PurePy

Language Specification, v0.27.12

Copyright © 2026 PurePy contributors. Distributed under the MIT License.

Contents

Summary **4**

1 Conformance **9**
1.1 Excluded Python features . . . 9
1.2 Conformance suite . . . 11
1.2.1 Attribution of ill-formedness . . . 11

2 Syntax **12**
2.1 Lexical syntax . . . 12
2.2 Contexts . . . 12
2.2.1 Context operations . . . 14
2.3 Statements and expressions . . . 14
2.3.1 Type annotations . . . 14
2.3.2 Operators . . . 17
2.4 Modules and programs . . . 17

3 Well-formedness **21**
3.1 Types . . . 21
3.1.1 Subtyping . . . 22
3.2 Expressions . . . 25
3.2.1 Operators . . . 28
3.3 Statements . . . 30
3.3.1 Function definitions . . . 34
3.3.2 Top-level statements . . . 35
3.4 Pattern matching . . . 36
3.4.1 Pattern checking . . . 36
3.4.2 Shapes and shape typing . . . 39
3.4.3 Matching . . . 39
3.5 Modules and programs . . . 42

4 Operational semantics **45**
4.1 Values and environments . . . 45
4.2 Auxiliary definitions . . . 46
4.2.1 Operators . . . 46
4.3 Expressions and statements . . . 48
4.4 Pattern matching . . . 57
4.5 Modules and programs . . . 57
4.6 Type soundness . . . 58

A Auxiliary functions (normative) **59**
A.1 Functions on syntax . . . 59
A.2 Functions on types and shapes . . . 62
A.3 Functions on classes and modules . . . 67
A.4 Functions on values . . . 70

B Excluded Python behaviours (informative) 78
B.1 Behaviours excluded by design . . . 78
B.1.1 Soft keywords as identifiers . . . 78
B.1.2 Mutable state . . . 78
B.1.3 Implicit coercions . . . 78
B.1.4 Load-dependent meaning . . . 78
B.1.5 Compatibility with mypy . . . 80
B.2 Features omitted from this version . . . 80
B.3 Runtime errors prevented by well-formedness . . . 80

Summary

This chapter collects the notation of the specification: the metavariables (Figure S1), the judgement forms with the figure defining each (Figure S2), and the auxiliary functions with the section defining each (Figures S3 and S4). The specification is mechanised in Isabelle/HOL[1]; where a figure, definition or result has a counterpart in the mechanisation, its caption or heading ends with the name of the counterpart, such as `check_program` in Figure 3.5–3.

[1] https://github.com/pure-py/isabelle-purepy

Names and literals		
x, y, z	variable	§2.2
f, c, α	variable; idiomatically f a function, c a class and α a type parameter	§2.2
ξ	optional type variable, α or ε	Figure 3.2–7a
q	name of a module, class, type alias or type parameter	Figure 2.3–1
n	number, an integer or a float; also sequence length and exit status, both integers	Figure 2.3–2
w	string	Figure 2.3–2
ℓ	literal	Figure 2.3–2
$\odot$, $\ominus$	binary and unary operator	Figure 2.3–5
Syntax		
e	expression	Figure 2.3–2
p	pattern	Figure 2.3–2
g	comprehension qualifier	Figure 2.3–2
b	case of a `match`	Figure 2.3–1
s	statement	Figure 2.3–1
d	function definition	Figure 2.3–1
τ, σ	type	Figure 2.3–3
ψ	type expression, a type as written	Figure 2.3–3
ν	primitive type	Figure 2.3–3
π	type or field scheme	Figure 2.3–3
ι	import	Figure 2.3–1
ϕ	field list of a dataclass	Figure 2.3–1
t	top-level statement	Figure 2.3–1
χ	type definition	Figure 2.3–1
u	argument: an expression, pattern or value	Chapter 2
a	argument list	Figure 2.3–2
Programs		
$\mathcal{M}$	program, a finite map from module names to bodies	§2.4
$\mathcal{P}$	predefined modules, each with its signature	§2.4
$\mathcal{A}$	command-line arguments of a run, a sequence of strings	§2.4
$\mathcal{E}(\mathcal{A})$	predefined modules, each with its environment, for a run with arguments $\mathcal{A}$	§2.4
Contexts		
Γ, Δ	context, a finite map from variables to context entries	Definition 2.1
θ	context entry	Definition 2.1
C, D	fully-qualified class name; also the class so named and the type of its instances	Definition 2.2
Σ	class table, a finite map from classes to class table entries	Definition 2.3
R	static outcome of a statement	§3.3
ζ	optional return type, τ or ε	Figure 3.3–1
Shapes		
k, l	shape, denoting values of a type, idiomatically l residual	Figure 3.4–3a
H	finite set of heads excluded by a shape	Figure 3.4–3a
h	head, a literal, class or list of length n	Figure 3.4–3a
β	bindings of a dictionary shape, a finite map from keys to shapes	Figure 3.4–3a
K, L	ordered sets of shapes, idiomatically K matched and L residual	Figure 3.4–3a
Values and results		
v	value	Figure 4.1–1
ρ	environment, a finite map from variables to values	Figure 4.1–1
$\mathcal{L}$	loaded module environments, a finite map from module names to environments	Figure 4.5–1
δ	dictionary entries	Figure 4.1–1
o	outcome of statement evaluation, a completion with printed values	Figure 4.1–1
ω	completion of a statement	Figure 4.1–1
r	result of expression evaluation	Figure 4.1–1
m	match result	Figure 4.1–1
κ	termination kind	Figure 4.1–1
f	meaning of a primitive, a function from a sequence of values to a result	Figure 4.1–1

Figure S1: Metavariables used throughout specification

Judgement	Reading	Defined in
Well-formedness		
$\Sigma \vdash \tau$ wf	type τ is well-formed	Figure 3.1–3a
$\Sigma \vdash \theta$ wf	context entry θ is well-formed	Figure 3.1–3b
$\Sigma \vdash \Gamma$ wf	context Γ is well-formed	Figure 3.1–3b
$\Sigma \vdash C$ wf-class	class C is well-formed	Figure 3.1–4a
$\vdash \Sigma$ wf	class table Σ is well-formed	Figure 3.1–4b
$\Sigma \vdash h : \tau$	head h has type τ	Figure 3.4–3b
$\Sigma \vdash k : \tau$	shape k has type τ	Figure 3.4–3c
$\Sigma; \Gamma \vdash k, p \rightsquigarrow_{\mathsf{h}} K, L$	shape k splits at the head tested by p, giving K and L	Figure 3.4–5
$\Sigma; \Gamma \vdash k, p \rightsquigarrow K, L$	pattern p matches shapes K of k, leaving L	Figure 3.4–4a
$\Sigma; \Gamma \vdash \vec{k}, \vec{p} \rightsquigarrow K, L$	patterns $\vec{p}$ match shapes K of $\vec{k}$, leaving L	Figure 3.4–4b
$\Sigma; \Gamma \vdash L, p \rightsquigarrow K, L'$	pattern p matches shapes K of L, leaving L'	Figure 3.4–4c
$\Sigma \vdash C[\vec{\sigma}] \asymp \tau$	$C[\vec{\sigma}]$ is the instance of C at τ	Figure 3.4–1a
$\Sigma \vdash C \parallel \tau$	class C is incomparable with τ	Figure 3.4–1b
$\tau \# \sigma$	τ and σ are apart	Figure 3.4–1c
$\Sigma \vdash \tau$ needs-eq ξ	equality at τ needs equality at type variable ξ, or at a callable type if ξ is ε	Figure 3.2–7a
$\Sigma \vdash C$ needs-eq ξ	equality at class C needs equality at its type parameter ξ, or at a callable type if ξ is ε	Figure 3.2–7b
$\Sigma; \Gamma \vdash p \lhd \tau \dashv \Delta$	pattern p checks against type τ, binding Δ	Figure 3.4–2
$\Gamma \vdash q : \theta$	name q denotes context entry θ	Figure 3.1–1
$\Sigma; \Gamma \vdash \psi \rhd \tau$	type expression ψ denotes type τ	Figure 3.1–2
$\Sigma \vdash \sigma <: \tau$	type σ is a subtype of τ	Figure 3.1–5
$\Sigma; \Gamma \vdash e \rhd \tau$	expression e synthesises type τ	Figure 3.2–1
$\Sigma; \Gamma \vdash e, \vec{e} \rhd \vec{\sigma}, \sigma$	callee e at arguments $\vec{e}$ synthesises parameter types $\vec{\sigma}$ and return type σ	Figure 3.2–4
$\Sigma; \Gamma \vdash e \lhd \tau$	expression e checks against type τ	Figure 3.2–5a
$\Sigma; \Gamma \vdash \vec{g} : \Delta$	comprehension qualifiers $\vec{g}$ are well-formed and bind Δ	Figure 3.2–5b
$\Sigma; \Gamma; \zeta \vdash s : R$	statement s has static outcome R, with return type ζ	Figure 3.3–1
$\Sigma; \Gamma \vdash d \rhd \pi$	definition d synthesises type scheme π	Figure 3.3–3a
$\Sigma; \Gamma \vdash d \lhd \pi$	definition d checks against type scheme π	Figure 3.3–3b
$\Sigma; \Gamma; \zeta \vdash \tau, L, \vec{b} : S, \tau', L'$	cases $\vec{b}$ have static outcome S, transforming remaining type τ and residual L to τ' and L'	Figure 3.3–4
$\Sigma; q; \Gamma \vdash t : R \dashv \Sigma'$	top-level statement t of module q has static outcome R, extending Σ to Σ'	Figure 3.3–5a
$\Sigma; \Gamma \vdash \chi$	type definition χ is well-formed	Figure 3.3–5b
$\Sigma \vdash \Gamma, x$ imports $\theta \dashv \Sigma'$	name x imported from module signature Γ has context entry θ	Figure 3.5–1
$\Sigma; q \vdash \vec{\imath} : \Gamma \dashv \Sigma'$	imports $\vec{\imath}$ of module q yield context Γ	Figure 3.5–2a
$\Sigma \vdash_{\mathsf{sub}} q, \Gamma, \vec{x} : \Gamma' \dashv \Sigma'$	submodule chain $\vec{x}$ extends signature Γ of module q to Γ'	Figure 3.5–2b
$\Sigma \vdash_{\mathsf{mod}} q : \Gamma \dashv \Sigma'$	module q checks with signature Γ	Figure 3.5–3a
$\vdash \mathcal{M}$	program $\mathcal{M}$ is well-formed	Figure 3.5–3b
Evaluation		
$\vdash v : \theta$	value v is well-typed at context entry θ	Figure 4.3–1a
$\vdash \rho : \Gamma$	environment ρ is well-typed at context Γ	Figure 4.3–1b
$\rho, s \Rightarrow o$	statement s evaluates to outcome o	Figure 4.3–2
$q, \rho, t \Rightarrow o$	top-level statement t of module q evaluates to outcome o	Figure 4.3–4
$\rho, e \Rightarrow r$	expression e evaluates to result r	Figure 4.3–5
$\rho, \vec{e} \Rightarrow r$	expressions $\vec{e}$ evaluate, left to right, to result r	Figure 4.3–8
$\rho, \vec{g} \Rightarrow r$	comprehension qualifiers $\vec{g}$ evaluate to result r	Figure 4.3–9
$q, \mathcal{L} \Rightarrow_{\mathsf{mod}} o$	body of module q evaluates to outcome o given loaded module environments $\mathcal{L}$	Figure 4.5–1a
$\vec{q}, \mathcal{L} \Rightarrow_{\mathsf{mod}} o$	bodies of modules $\vec{q}$ evaluate to outcome o given loaded module environments $\mathcal{L}$	Figure 4.5–1b
$\mathcal{M}, \mathcal{A} \Rightarrow \vec{v}, \kappa$	program $\mathcal{M}$ run with arguments $\mathcal{A}$ prints $\vec{v}$ and ends with termination kind κ	Figure 4.5–1c

Figure S2: Judgement forms

Function	Result	Meaning	Section
Functions on syntax			
assigns(t)	set of variables	variables declared or assigned in t, not descending into nested functions	§A.1
declares(t), declares($\vec{\imath}$)	sequence of variables	names declared in t, or bound by imports $\vec{\imath}$, in order and with duplicates, not descending into nested functions	§A.1
type-entry$_\Gamma(q, \vec{\chi}, i)$, type-entry$_\Gamma(q, \vec{\chi})$	context entry, or sequence of context entries	context entry associated with definition χ_i of mutual type region $\vec{\chi}$ of module q under Γ, or with each definition of $\vec{\chi}$	§A.1
type-context$_\Gamma(q, \vec{\chi}, i)$	context	context resolving type expressions of definition χ_i of mutual type region $\vec{\chi}$ of module q under Γ	§A.1
class-table$_\Gamma(q, \vec{\chi}, i)$, class-table$_\Gamma(q, \vec{\chi})$	class table	class table contributed by definition χ_i, or by all definitions, of mutual type region $\vec{\chi}$ of module q under Γ	§A.1
scope($\vec{y}$, s)	context	each variable declared or assigned in s unbound; defined iff parameters or imported names $\vec{y}$, names declared in s, and pattern variables of s are distinct and pairwise disjoint	§A.1
binds(p), binds($\vec{p}$), binds($\vec{g}$), binds(s)	set of variables	variables bound by pattern p, by patterns $\vec{p}$, by generators of qualifiers $\vec{g}$, or by patterns within statement s	§A.1
captures(e)	set of variables	variables of enclosing scope captured by lambdas in e	§A.1
captures($\vec{g}$)	set of variables	variables of enclosing scope captured by lambdas in qualifiers $\vec{g}$	§A.1
fv(e), fv($\vec{e}$), fv(τ)	set of variables	free variables of expression e, or of each expression in $\vec{e}$; type variables of type τ	§A.1
tyvars(Γ)	set of variables	type variables bound by context Γ or free in its entries	§A.1
Functions on types and shapes			
base-type(ℓ), base-type(τ)	type	base type of literal ℓ, or of type τ	§A.2
$\sigma \vee_\Sigma \tau$, $\bigvee_\Sigma(\vec{\tau})$	type	larger type if comparable, and union otherwise	§A.2
$\sigma \wedge_\Sigma \tau$	type	greatest type below both	§A.2
match-args(τ, σ), match-args($\vec{\tau}, \vec{\sigma}$)	context binding type variables at types	type arguments from one-way matching of τ against σ, or of $\vec{\tau}$ against $\vec{\sigma}$	§A.2
type-args$_\Sigma(\vec{\alpha}, \sigma, \tau)$, type-args$_\Sigma(\vec{\alpha}, \vec{\sigma}, \vec{\tau})$	context binding type variables at types	type arguments from one-way unification of σ with τ, or of $\vec{\sigma}$ with $\vec{\tau}$	§A.2
elem-type$_\Sigma(\tau)$	type	type of elements drawn from τ by generator	§A.2
shapes$_\Sigma(\tau, H)$, shapes$_\Sigma(\vec{\tau})$	ordered set of shapes or of shape sequences	shapes of τ once heads in H are excluded; sequences of shapes of $\vec{\tau}$	§A.2
shape-type(k)	type	least type of shape k	§A.2
matched-as$_\Sigma(\Gamma, \tau, p)$	type	type τ is matched as by pattern p	§A.2
remaining$_\Sigma(\Gamma, \tau, p)$	type	type remaining from τ if pattern p fails to match	§A.2
resolve-op$_\Sigma(\odot, \sigma_1, \sigma_2)$, resolve-op$_\Sigma(\ominus, \sigma)$	type	return type of most specific overload at the operand types	§A.2

Figure S3: Auxiliary functions: syntactic analyses and types

Function	Result	Meaning	Section
Functions on classes and modules			
submods(q)	context	stub for each submodule of q	§A.3
unique-to(q', q)	sequence of module names	modules which strictly contain q' but do not contain q, shortest first	§A.3
loads($\vec{q}$, $\vec{q}'$)	sequence of module names	modules $\vec{q}$ extended by modules loaded by $\vec{q}'$, in load order	§A.3
import-env$_{\mathcal{L}}$($\vec{\imath}$)	environment	environment bound by imports $\vec{\imath}$ given loaded module environments $\mathcal{L}$	§A.3
own-fields(ϕ), own-fields(χ)	sequence of bindings	fields declared by ϕ or χ, with type expressions	§A.3
type-params$_\Sigma$(C)	sequence of type parameters	type parameters of class C	§A.3
base$_\Sigma$(C)	instantiated class or $\bot$	base class of C at its declared type arguments	§A.3
fields$_\Sigma$(C)	field scheme	fields of class C with their types, inherited first, under its type parameters	§A.3
ancestors$_\Sigma$(C)	set of classes	C and its base classes	§A.3
ancestor-args$_\Sigma$($C[\vec{\tau}]$, D)	sequence of types	type arguments of instantiation of D above $C[\vec{\tau}]$	§A.3
field-map$_\Sigma$(C, $\vec{u}$)	map from variables	argument supplied for each field of C; defined iff arguments cover them exactly	§A.3
resolve(ρ, q)	value	value denoted by name q in ρ	§A.3
name(q)	string	string literal naming module q	§A.3
Functions on values			
type-of(v)	type	run-time type of v	§A.4
real(n)	extended real number	denotation of number n other than nan	§A.4
fl(x)	float	real number x rounded to float	§A.4
float(x)	result	real number x rounded to float; aborts(OverflowError) if infinite	§A.4
trunc(x)	integer	integer part of real number x	§A.4
floordiv(n, n')	float	floor division of floats	§A.4
mod(n, n')	float	remainder of float division, with sign of n'	§A.4
eq$_\Sigma$(v, v')	result	structural equality; aborts(undef) between functions, between Boolean and number, and between nan and nan in container	§A.4
eq-elems$_\Sigma$($\vec{v}$, $\vec{v}'$)	result	elementwise equality of value sequences	§A.4
in$_\Sigma$(v, v')	result	whether v is element of v'; excluded for range or non-string key, TypeError otherwise	§A.4
in-elems$_\Sigma$(v, $\vec{v}$)	result	whether $\vec{v}$ has element equal to v	§A.4
elems(v)	sequence of values	elements of list, tuple or string	§A.4
iter(v)	sequence of values	values drawn from v by generator: keys of dictionary, $m, \ldots, n-1$ for range with start m and stop n	§A.4
getitem(v, v')	result	subscripting; IndexError for index out of range, KeyError for missing key, excluded for range, Boolean index or non-string key, TypeError otherwise	§A.4
result(o)	result	statement outcome as result: returned value, `None` for normal completion, abort unchanged	§A.4
match$_\Sigma$(ρ, p, v)	match result	bindings of pattern variables if p matches v, no-match if not, undef if equality of numeric literal pattern p and v undefined, excluded for list or tuple pattern against the other or a range, match-fields$_\Sigma$ for constructor pattern	§A.4
match-fields$_\Sigma$(ρ, C, a, $\vec{z}:\vec{v}$)	match result	sub-patterns of a against object fields; TypeError for excess sub-patterns, excluded if fields not covered	§A.4
dispatch(ρ, v, $\vec{p}$, $\vec{s}$)	bindings and statement	case taken by `match`, or termination kind if matching aborts	§A.4
update(δ, δ')	dictionary entries	δ updated by δ' in order, later replacing earlier	§A.4
setitem(δ, w, v)	dictionary entries	δ with w bound to v, replacing earlier binding	§A.4

Figure S4: Auxiliary functions: classes, modules and values

1 Conformance

This document defines PurePy version 0.27.12. PurePy is a subset of Python, defined by static rules: a Python program is *a PurePy program* if it has the syntax of Chapter 2, and a PurePy program is *well-formed* if it satisfies the well-formedness rules of Chapter 3. Not every behaviour excluded by PurePy can be rejected statically however (the excluded behaviours are summarised in informative Annex B). A run which reaches one of these *dynamically excluded* operations, such as comparison of two functions, has undefined behaviour, and aborts with undef. In all other respects PurePy agrees with Python: every well-formed program is *syntactically valid*, in that it does not raise `SyntaxError` under Python, and every terminating run with defined behaviour under PurePy produces the same result under Python. Figure 1–1 classifies programs and runs by their outcomes under the two languages. The hatched cells of the figure are empty because every well-formed program is syntactically valid. Each result in the last column is the same as under Python. For PurePy 0.27.12, Python means Python 3.12 and mypy means mypy 2.3.1 with its default options. Chapter 2 gives the abstract syntax; the lexical syntax is that of Python (§2.1). Chapter 3 defines well-formedness, the static part of the definition, and Chapter 4 the operational semantics, which is defined for well-formed programs.

A language implementation conforms to the PurePy specification if, on every run of a well-formed program with defined behaviour, it produces the Python result; on any other run it may either abort and report undef, following the semantics, or produce the Python result.* The conformance suite (§1.2) is organised by outcome under Python and PurePy (Figure 1–1).

[**DAO:** not sure I get this last bit]

	PurePy rejects				**PurePy accepts**	
	Static			Dynamic		
Python	*parser*	*type checker*	*not yet*	aborts(undef)	*defined*	*Total*
`SyntaxError`	**17**	**2**	**1**			20
Other exception	**4**	**100**	**3**	**3**	**29**	139
No exception	**29**	**311**	**74**	**7**	**316**	737
Total	50	413	78	10	345	896

Python error prevented · Excluded by design · Same result as Python
Cannot occur · Omitted from this version · Well typed under mypy

Figure 1–1: Conformance tests, by outcome under Python and PurePy

1.1 Excluded Python features

A syntactically valid Python program may use an *excluded Python feature*: a syntactic form, module or attribute which PurePy does not include. Each excluded feature is either *excluded by design*, because it conflicts with the design goals of PurePy, such as immutability, or *omitted from this version*, meaning it is likely to appear in a future version of the language. Figure 1.1–1 lists the features excluded by design; any other excluded feature is merely omitted from this version, and Figure B.2–1 in informative Annex B lists the common ones.

A program may also be rejected because it uses an included feature in a way which the well-formedness rules do not allow, such as a non-Boolean test. Such uses are excluded by design, except for the uses of predefined functions listed in Figure 1.1–2, each of which depends on a feature omitted from this version. Every dynamically excluded operation is excluded by design.

Feature	Python	Exclusion
Mutable state		
`global` and `nonlocal`	Assign to variable of enclosing scope	Syntactic
`del`	Remove binding	Syntactic
Augmented assignment `+=`, etc.	Update in place if type supports it	Syntactic
Walrus operator `:=`	Assign within expression	Syntactic
Item assignment `xs[0] = v`	Update container in place	Syntactic
Mutating methods of `list` and `dict`, such as `append` and `update`	Update container in place	Static
Attribute assignment `p.x = v`	Update object in place	Syntactic
Built-ins `setattr` and `delattr`	Set or delete attribute named by string	Static
`is` and `is not`, except with `None`, `True` or `False`	Compare objects by identity	Syntactic
Built-in `id`	Return identity of object	Static
Iteration		
`for` and `while` loops, `break` and `continue`	Iterate by updating loop variable	Syntactic
`yield`, `yield from` and generator expressions	Create generator, consumed by mutation	Syntactic
Built-ins `iter` and `next`	Create or advance iterator	Static
Load-dependent meaning		
Import inside function body or branch, or after statement	Dependencies of module discovered as body executes	Syntactic
Wildcard import `from q import *`	Introduce names of `q` implicitly	Syntactic
Reflection		
Built-ins `eval` and `exec`	Run code given as string	Static
Built-in `getattr`	Read attribute named by string	Static
Built-ins `globals`, `locals` and `vars`	Return namespace as dictionary	Static
Built-in `__import__`	Import module named by string	Static
Implicit coercions		
Dictionary with keys other than strings, such as `{1: v}`	Accept any hashable key	Static
Key type other than `str`, such as `dict[int, str]`	Accept any hashable key	Syntactic
Optional and gradual typing		
Parameter or return type without annotation	Accept definition without annotations	Syntactic
Annotation which is not a type, such as `1` or `list[int, int]`	Ignore annotation	Syntactic
`Any` and `cast` of `typing`	Disable static checking of value	Static
Reserved words		
Soft keywords `match`, `case`, `type` and `_` as identifiers	Allow as identifiers except where they are keywords	Syntactic

Figure 1.1–1: Python features excluded by design

Use	Depends on excluded feature	Exclusion
`print` inside function	Expression-level printing	Static
`print` as first-class value	`*args`, keyword-only parameters and default values	Static
`round` with second argument	Default values	Static
`sys.exit` with no arguments	Default values	Static

Figure 1.1–2: Uses of predefined functions omitted from this version

1.2 Conformance suite

The conformance suite pairs each test with the verdict given by Python itself, and runs on CPython, GraalPy, and Pyodide. A test in `python-raises/typing-rejects/`, for example, is a program which raises an exception other than `SyntaxError` under Python and is rejected by the PurePy type checker; the directory of a test names the row of Figure 1–1 for the outcome under Python, followed by the column for the outcome under PurePy. A test which fails because of a bug reported in mypy or a Python implementation is recorded as expected to fail; Figure 1.2–1 lists these behaviours.

Construct	Reported in	Behaviour	Issue
Match on Boolean literal scrutinee, with cases covering its value	mypy	Reports missing return, ignoring Boolean value known from literals	python/mypy#22202
Operator `//` on floats	GraalPy	Quotient 1 greater than in Python for some operands, such as `1.0 // 0.1`	oracle/graalpython#1274

Figure 1.2–1: Behaviours reported as bugs, with disagreement accepted by conformance suite

1.2.1 Attribution of ill-formedness

So that rejections can be traced to the specification, tests of ill-formed programs attribute the ill-formedness to the judgement at which each program fails and, if possible, to a rule. Implementations can then compare diagnostics with the attributions. The first line of the main module of each test cites the judgement form, by the identifier exported for the form by the conformance package, followed optionally by a rule:

```
# judgement: synth, rule: call
```

The cited judgement is the innermost judgement to fail in an attempted derivation. For example, a call with too few arguments to a callee of union type cites call rather than call-union, because the call fails for one disjunct of the callee type.

The conclusion of a rule *matches* a judgement if a substitution for the metavariables of the conclusion yields the judgement, apart from any derived type, context or value, which is absent from a failing judgement. A judgement *selects* a rule if the rule is the only rule with a conclusion matching the judgement, or, of several rules with matching conclusions, the only rule with its *selecting premises* holding. Selecting premises are the premises which decide between rules with matching conclusions, such as premises on the types synthesised by subexpressions or on the context entries of names. Citations name the rule selected by the cited judgement, or the judgement alone if the judgement does not select a rule. For example, an unbound variable cites var, which is the only rule for variables, and `1 + "a"` cites binop, which has operator resolution as a side condition. A class given too many type arguments cites ty-class, which is selected by the context entry of the class name, and a list subscripted by a string cites subscript-list, which is selected by the type of the list. An annotation with an unbound name cites the judgement for type expressions alone, and a subscripted integer cites the synthesis judgement alone, since the selecting premises of every matching rule fail.

Type mismatches cite the subtyping judgement between the synthesised and expected types, and the rule selected by the subtyping judgement. In the subtyping judgement, a literal type is taken as its base type unless the other type is also a literal type. Thus an expression of type `list[str]` assigned to a variable declared at `list[int]` cites subty-list. If the subtyping judgement does not select a rule, the type mismatch cites instead the judgement and rule with the subtyping judgement as a premise, so the declaration x`: int = "a"` cites declare-assign rather than check-syn.

2 Syntax

The syntax of PurePy is given in Figures 2.3–1, 2.3–2 and 2.3–3. Figure S1 summarises the metavariables used throughout. A sequence $\vec{x}$ denotes $x_1, \dots, x_n$ for some n; inside the brackets of a tuple, list, type application or argument list it stands for the comma-separated elements, so $(\vec{e})$ is $(e_1, \dots, e_n)$. The sequence $x \cdot \vec{x}$ has head x and tail $\vec{x}$, the sequence $\vec{x} \cdot x$ has last element x, $\vec{x} \mathbin{+\!\!+} \vec{y}$ is concatenation, $\mathbin{+\!\!+}(\vec{x})$ is the concatenation of the sequences in $\vec{x}$, and x^n is x repeated n times. Comma-separated elements inside set braces are pairwise distinct, so $\{x_1, \dots, x_n\}$ has n elements; the set of the elements of a sequence, $\{\vec{x}\}$, assumes no distinctness. A sequence comprehension $(x_i \mid i \in I)$, where the finite index set I carries a total order, denotes the sequence of the x_i in that order. Two sequences of equal length paired by a colon, $\vec{x} : \vec{\psi}$, denote the sequence of pairs $x_1 : \psi_1, \dots, x_n : \psi_n$. A judgement applied to sequences of equal length holds pointwise, so that $\Sigma; \Gamma \vdash \vec{\psi} \rhd \vec{\tau}$ abbreviates $\Sigma; \Gamma \vdash \psi_i \rhd \tau_i$ for each i. When the order of its elements matters, a set $\{x_1, \dots, x_n\}$ is *ordered* by position. A comprehension over an ordered set preserves the order of its source; $X \setminus Y$ keeps the surviving elements of X in their order; $X \uplus Y$, defined when X and Y are disjoint, appends the elements of Y to those of X; $X \cup Y$ appends the elements of Y not in X; and a product of ordered sets is ordered lexicographically. The brackets $\lfloor x \rfloor$ and $\lceil x \rceil$ denote the floor and ceiling of a real number x.

2.1 Lexical syntax

The grammar of PurePy is an abstract syntax, so its metavariables n and w range over numbers and strings rather than over the numerals and string literals of the source text. Numbers are integers or floats, which are distinct even when they denote the same real number, such as `3` and `3.0`. Floats are IEEE 754 double-precision numbers, and float literals are rounded to the nearest float.[1] The constants inf and −inf are the infinities of IEEE 754, and the constant nan is a NaN of IEEE 754; every operation with a NaN result gives nan, so that nan is the only NaN of PurePy and evaluation is deterministic. Variables are Python identifiers other than keywords and the soft keywords `match`, `case`, `type` and _ (§B.1.1); _ is the wildcard pattern. Python writes a negative number as unary minus applied to a numeric literal; the translation to abstract syntax folds the minus into the literal, so a number may be negative. Python's grammar writes a chain of `and` or `or` operators as one operator applied to a list of operands; the translation groups the chain to the left as binary operators. The translation of source text into abstract syntax otherwise maps each form of Python's grammar to the form of the same shape in Figures 2.3–1 and 2.3–2, including the indentation that delimits blocks. The translation depends on the source text alone. In particular, a call $q(a)$ or $q[e](a)$ is translated in the same way whether or not q denotes a class; the well-formedness rules treat such a call as a constructor call when q denotes a class (Chapter 3). A list of type parameters is always present in the abstract syntax and may be empty; in source an empty list is omitted, because in Python's grammar the list is optional and, when present, has at least one element. An expression statement `print`(e) maps to PurePy's print statement. The grammar as written admits some programs which Python rejects as syntax errors, such as a function with two parameters of the same name or a `return` outside a function. Figure 2.3–4 lists the conditions, most of them distinctness conditions, which rule out these programs.

2.2 Contexts

Metavariables x, y, z, f, c and α range over *variables*; idiomatically f is used when the variable is expected to denote a function, c when it is expected to denote a class and α when it is a type parameter. A *name* q is a non-empty sequence of variables separated by dots (Figure 2.3–1): a variable is a *simple name*, and a name with

[1] The Python language reference requires only machine double precision for floats (`https://docs.python.org/3/reference/datamodel.html`). CPython, PyPy and GraalPy round float literals to the nearest double, and so meet the stronger requirement of PurePy.

at least one dot is *qualified*. A name is *fully qualified* when it is read from the root of the module hierarchy; modules (§2.4) and classes (Definition 2.2) are identified by fully-qualified name.

Context entries			
θ	::=	τ	(Definitely assigned)
	|	$\mathsf{DU}(\tau)$	(Definitely unassigned)
	|	$\mathsf{PU}(\tau)$	(Possibly unassigned)
	|	$\mathsf{Unbound}$	(Unbound)
	|	$\forall\vec{\alpha}.\,\tau$	(Type scheme, $\vec{\alpha}$ non-empty)
	|	$\mathsf{PU}(\forall\vec{\alpha}.\,\tau)$	(Possibly unassigned type scheme)
	|	$\mathsf{Mod}(q)$	(Module stub)
	|	$\mathsf{Mod}(q,\Gamma)$	(Checked module, with signature Γ)
	|	$\mathsf{Type}(C)$	(Class)
	|	$\mathsf{TypeVar}$	(Type parameter)
	|	$\mathsf{TypeAlias}_\Gamma(q,\vec{\chi},i)$	(Type alias)
	|	Predef	(Predefined name)
Class entries			
		$\mathsf{ClassDef}_\Gamma(q,\vec{\chi},i)$	(Class definition)

Figure 2.2–1: Context entries and class table entries

Definition 2.1 (Context)**.** A context Γ (or Δ) is a finite map from variables to *context entries* θ (Figure 2.2–1). We write $\Gamma, x : \theta$ for Γ with its entry for x replaced by θ, $\mathrm{dom}(\Gamma)$ for the set of variables with an entry in Γ, and $\emptyset$ for the empty context.

A predefined name is a member of a predefined module whose meaning is fixed by PurePy, such as `list` or `dataclass`; its entry records only that it is in scope, and it does not denote a value, so it cannot appear in an expression. A type parameter or type alias denotes a type rather than a value, so it too can be mentioned only by a type expression. The entries $\mathsf{Type}(C)$, $\mathsf{TypeVar}$ and $\mathsf{TypeAlias}_\Gamma(q,\vec{\chi},i)$ are named after the Python classes `type`, `typing.TypeVar` and `typing.TypeAliasType` of the objects denoted by a class name, type parameter or alias at run time. The entry $\mathsf{TypeAlias}_\Gamma(q,\vec{\chi},i)$ records the alias declared by χ_i in the mutual type region $\vec{\chi}$ of module q, resolved under Γ.

The definite-assignment terminology is borrowed from Java [Gosling et al., 2025, §16]: a variable bound to a type is *definitely assigned*. A variable with entry $\mathsf{DU}(\tau)$ is declared at τ and *definitely unassigned*, like a blank final in Java before its assignment. A variable with entry $\mathsf{PU}(\tau)$ is declared at τ and *possibly unassigned*, because it is assigned on some paths but not on all. A function name with entry $\mathsf{PU}(\forall\vec{\alpha}.\,\tau)$ is bound at the type scheme $\forall\vec{\alpha}.\,\tau$ on some paths but not on all. A variable with entry $\mathsf{Unbound}$ is declared later in its scope, which Python reports at runtime as an unbound local. It is useful to represent contexts as total functions on the set of all variables by extending the codomain with $\bot$ to indicate *undefined*.

Definition 2.2 (Class name)**.** *Class names* C are fully-qualified names $q.c$, where c is the name under which the class is defined in module q. A module defines each name at most once, so class names identify classes, and we write "class C" for the class named C. A class name is also the type of the instances of the class.

Definition 2.3 (Class table)**.** A *class table* Σ maps each class name C in its domain to a *class table entry* $\mathsf{ClassDef}_\Gamma(q,\vec{\chi},i)$ (Figure 2.2–1), which records the class definition χ_i of mutual type region $\vec{\chi}$ of module q defining C, and the context Γ in which the region appears. Every class named in Σ is in $\mathrm{dom}(\Sigma)$, and the base-class relation on $\mathrm{dom}(\Sigma)$ is acyclic.

The type parameters, fields and base class of C are read from $\Sigma(C)$ by auxiliary functions $\text{type-params}_\Sigma$, fields_Σ and base_Σ (§A.3), the latter two resolving the field annotations and base class of the definition under the context given by $\text{type-context}_\Gamma(q,\vec{\chi},i)$ (§A.1). The class table is similar to the class table of Featherweight Java [Igarashi et al., 2001]; the well-formedness rules build it while checking the class definitions of the program (Figure 3.3–5).

2.2.1 Context operations

Contexts admit a non-commutative monoid of sequential composition $\Gamma \otimes \Delta$ ("override") and up to type equivalence, an idempotent commutative semigroup of parallel composition $\Gamma \oplus_\Sigma \Delta$ ("merge"). Together these form a *right near-semiring* [van Hoorn and van Rootselaar, 1967]: $\otimes$ right-distributes over $\oplus$, but left distributivity does not hold. Both are defined pointwise, by $(\Gamma \cdot \Delta)(x) = \Gamma(x) \cdot \Delta(x)$ for each operation $\cdot$, from operations on context entries. Override is right-biased and has $\bot$ as both left and right unit, and is defined in the same way on any finite map with $\bot$ for undefined, such as an environment or class table. Merge has no unit, since a variable defined in only one of the two contexts is possibly unassigned after the merge. Merge of different type schemes is undefined; a scope declares each name at most once, so the type schemes of functions agree across branches and the case does not arise. The types of declared variables also agree across branches, but patterns bind variables without declared types, at types which may differ between branches. When both entries of a variable are defined, merge therefore gives an entry at the join of the two types, definitely assigned if both entries are definitely assigned and possibly unassigned otherwise.

Definition 2.4 (Override of context entries)**.**

$$\begin{aligned} \bot \otimes \theta &= \theta \\ \theta \otimes \bot &= \theta \\ \theta \otimes \theta' &= \theta' \quad \text{(otherwise)} \end{aligned}$$

Definition 2.5 (Merge of context entries)**.**

$$\begin{array}{lll} \theta \oplus_\Sigma \theta = \theta & & \\ \sigma \oplus_\Sigma \tau = \sigma \vee_\Sigma \tau & \text{if } \sigma \neq \tau & \\ \theta \oplus_\Sigma \theta' = \mathsf{PU}(\sigma \vee_\Sigma \tau) & \text{if } \theta \neq \theta' \text{ and } \theta \in \{\mathsf{DU}(\sigma), \mathsf{PU}(\sigma)\}, \theta' \in \{\tau, \mathsf{DU}(\tau), \mathsf{PU}(\tau)\} & \\ \sigma \oplus_\Sigma \theta = \mathsf{PU}(\sigma \vee_\Sigma \tau) & \text{if } \theta \in \{\mathsf{DU}(\tau), \mathsf{PU}(\tau)\} & \\ \bot \oplus_\Sigma \theta = \theta \oplus_\Sigma \bot = \mathsf{PU}(\tau) & \text{if } \theta \in \{\tau, \mathsf{DU}(\tau), \mathsf{PU}(\tau)\} & \\ \theta \oplus_\Sigma \theta' = \mathsf{PU}(\forall \vec{\alpha}.\, \tau) & \text{if } \theta \neq \theta' \in \{\forall \vec{\alpha}.\, \tau, \mathsf{PU}(\forall \vec{\alpha}.\, \tau), \bot\} & \end{array}$$

2.3 Statements and expressions

Statements are sequenced by juxtaposition, $s\ s'$; a function body, branch of a conditional and body of a case are statements. *Type definitions* χ are class definitions or `type` statements. *Top-level statements* t are statements or mutual type regions, also sequenced by juxtaposition; the body of a module after its imports is a top-level statement, so type definitions occur only at the top level of a module. *Type aliases* are declared by `type` statements. Function definitions, class definitions and `type` statements bind distinct *type parameters* $\vec{\alpha}$, written in brackets after the name as in Python 3.12. A *mutual def region* $d_1 \ldots d_{n+1}$ is a maximal contiguous block of `def` statements: any other statement separates mutual def regions. Maximality is not enforced by the grammar in Figure 2.3–1; the parser is responsible for grouping consecutive `def`s into a single mutual def region. A *mutual type region* $\chi_1 \ldots \chi_{n+1}$ is a maximal contiguous block of type definitions, grouped by the parser in the same way. The well-formedness rules treat each mutual def region and each mutual type region as one mutually-recursive binding.

An *argument list* a (Figure 2.3–2) is a sequence of positional arguments followed by keyword arguments $x = u$, where an argument u is an expression in a call and a pattern in a constructor pattern. A constructor call may also pass type arguments to the class; a constructor pattern cannot, since Python does not permit type arguments in a constructor pattern. The auxiliary function field-map$_\Sigma$ (§A.3) pairs each field of the class with an argument: the positional arguments fill the leading fields in order, and a keyword argument $x = u$ fills the field x. It is defined only if every field receives exactly one argument.

2.3.1 Type annotations

Parameters, returns and dataclass fields are annotated with type expressions ψ, and an assignment may be annotated. A type expression is an expression (Figure 2.3–2) denoting a type τ (Figure 2.3–3). Type expressions

Statements			
s	::=	$s\ s'$	(Seq)
	|	`pass`	(Pass)
	|	x`:` ψ	(Declare)
	|	$x = e$	(Assign)
	|	x`:` $\psi = e$	(DeclareAssign)
	|	e	(ExprStmt)
	|	`print(`e`)`	(Print)
	|	`assert` e	(Assert)
	|	`assert` e, e'	(AssertMsg)
	|	`return`	(ReturnNone)
	|	`return` e	(Return)
	|	`if` e `:` s `elif` e_1 `:` s_1 … `elif` e_n `:` s_n	(If)
	|	`if` e `:` s `elif` e_1 `:` s_1 … `elif` e_n `:` s_n `else :` s'	(IfElse)
	|	`match` e`:` b_1 … b_{n+1}	(Match)
	|	d_1 … d_{n+1}	(MutualDefRegion)
Top-level statements			
t	::=	s	(Stmt)
	|	$t\ t'$	(TopSeq)
	|	χ_1 … χ_{n+1}	(MutualTypeRegion)
Type definitions			
χ	::=	`@dataclass class` $c[\vec{\alpha}](q[\vec{\psi}])$ `:` ϕ	(DataclassExtend)
	|	`@dataclass class` $c[\vec{\alpha}]$ `:` ϕ	(Dataclass)
	|	`type` $x[\vec{\alpha}] = \psi$	(TypeAlias)
Imports			
ι	::=	`import` q	(PlainImport)
	|	`from` q `import` $x_1, \ldots, x_{n+1}$	(FromImport)
Cases			
b	::=	`case` p`:` s	(Case)
Field lists			
ϕ	::=	x_1`:` ψ_1 … x_{n+1}`:` ψ_{n+1}	(FieldList)
	|	`pass`	(EmptyFieldList)
Names			
q	::=	$x_1.\cdots.x_{n+1}$	(Name)
Function definitions			
d	::=	`def` $f[\vec{\alpha}](\vec{x}$`:` $\vec{\psi})$ `->` ψ`:` s	(Def)

Figure 2.3–1: Modules and statements

and types have the same forms, except that a type expression has a type-name form $q[\vec{\psi}]$, in place of which a type has two forms: *instantiated classes* $C[\vec{\tau}]$, which are classes applied to type arguments, and type variables α. A primitive type is a name, or the literal `None`; `list`, `tuple`, `dict`, `Callable` and `Literal` are names. Following Python's typing module, a single type argument stands for the one-element tuple; when there are no type arguments the subscript is omitted, unless the type takes a variable number of type arguments, in which case the empty tuple must be supplied. The only such type is `tuple`, so the empty tuple type is `tuple[()]`. The parameter list of a `Callable` type is a list. In a literal type `Literal`$[\ell]$, the literal ℓ may not be a float; floats are excluded from literals by the Python typing specification.[2] A union $\psi \mid \psi'$ applies the binary operator `|`. A dataclass specifies its base class by a type expression of the form $q[\vec{\psi}]$. Resolution (§3.1) takes $q[\vec{\psi}]$ to $C[\vec{\tau}]$

[2] https://typing.python.org/en/latest/spec/literal.html

Expressions			
e	::=	x	(Var)
	|	ℓ	(Literal)
	|	$e(a)$	(Call)
	|	$\{e_1 : e'_1, \ldots, e_n : e'_n\}$	(Dict)
	|	$e.x$	(Attribute)
	|	$e[e']$	(Subscript)
	|	$e \odot e'$	(BinOp)
	|	$\ominus e$	(UnOp)
	|	e `and` e'	(And)
	|	e `or` e'	(Or)
	|	e_1 `if` e `else` e_2	(CondExpr)
	|	$[e_1, \ldots, e_n]$	(List)
	|	$e_1, \ldots, e_n$	(Tuple)
	|	$[\, e\ g_1\ \ldots\ g_n]$	(ListComp)
	|	$\{e : e'\ g_1\ \ldots\ g_n\}$	(DictComp)
	|	`lambda` $\vec{x}$: e	(Lambda)

Comprehension qualifiers			
g	::=	`if` e	(Guard)
	|	`for` x `in` e	(Generator)

Literals			
ℓ	::=	n	(Number)
	|	w	(String)
	|	`True`	(True)
	|	`False`	(False)
	|	`None`	(None)

Argument lists			
a	::=	$\vec{u}, \vec{x}{=}\vec{u}'$	(Args)

Patterns			
p	::=	ℓ	(PatLiteral)
	|	x	(PatVar)
	|	_	(PatWildcard)
	|	$[p_1, \ldots, p_n]$	(PatList)
	|	$p_1, \ldots, p_n$	(PatTuple)
	|	$q(a)$	(PatConstr)
	|	$\{\vec{w} : \vec{p}\}$	(PatDict)
	|	p `as` x	(PatAs)

Figure 2.3–2: Expressions and patterns

when q denotes the class C, to the type variable α when q is a type parameter α and there are no arguments, and to the instantiation of the alias when q denotes a type alias. The substitution $[\vec{\tau}/\vec{\alpha}]\sigma$ replaces each occurrence of α_i in σ by τ_i, a plain replacement, since every type variable in a type is free, and is defined only if $\vec{\tau}$ and $\vec{\alpha}$ have the same length. A *scheme* π (Figure 2.3–3) binds type parameters $\vec{\alpha}$ in its body. Schemes are identified up to renaming of their type parameters. The body of a *type scheme* $\forall\vec{\alpha}.\ \tau$ is a type; the body of a *field scheme* $\forall\vec{\alpha}.\ \vec{x} : \vec{\tau}$ is a sequence of fields $\vec{x}$ with their types $\vec{\tau}$. The *instantiation* of π at $\vec{\sigma}$, written $\pi[\vec{\sigma}]$, substitutes $\vec{\sigma}$ for $\vec{\alpha}$ in the body: $[\vec{\sigma}/\vec{\alpha}]\tau$ for a type scheme and $\vec{x} : [\vec{\sigma}/\vec{\alpha}]\vec{\tau}$ for a field scheme. A scheme without type parameters is its body, with the $\forall\vec{\alpha}$ omitted. The fields of a class (§A.3) form a field scheme.

Primitive types			
ν	::=	`object`	(Object)
	|	`Never`	(Never)
	|	`None`	(None)
	|	`bool`	(Bool)
	|	`int`	(Int)
	|	`float`	(Float)
	|	`str`	(Str)
	|	`Sized`	(Sized)

Type expressions			
ψ	::=	ν	(Primitive)
	|	`list`$[\psi]$	(List)
	|	`tuple`$[\psi_1, \ldots, \psi_n]$	(Tuple)
	|	`dict`$[$`str`$, \psi]$	(Dict)
	|	`Callable`$[[\psi_1, \ldots, \psi_n], \psi]$	(Callable)
	|	`Literal`$[\ell]$	(Literal, ℓ not a float literal)
	|	$q[\psi_1, \ldots, \psi_n]$	(TypeName)
	|	$\psi \mid \psi'$	(Union)

Types			
τ	::=	ν	(Primitive)
	|	`list`$[\tau]$	(List)
	|	`tuple`$[\tau_1, \ldots, \tau_n]$	(Tuple)
	|	`dict`$[$`str`$, \tau]$	(Dict)
	|	`Callable`$[[\tau_1, \ldots, \tau_n], \tau]$	(Callable)
	|	`Literal`$[\ell]$	(Literal, ℓ not a float literal)
	|	α	(TypeVar)
	|	$C[\tau_1, \ldots, \tau_n]$	(Class)
	|	$\sigma \mid \tau$	(Union)

Schemes			
π	::=	$\forall \vec{\alpha}.\ \tau$	(Type scheme)
	|	$\forall \vec{\alpha}.\ \vec{x} : \vec{\tau}$	(Field scheme)

Figure 2.3–3: Type expressions, types and schemes

Form	*Condition*
`@dataclass class` $c[\vec{\alpha}](q[\vec{\psi}])$ `:` ϕ, `@dataclass class` $c[\vec{\alpha}]$ `:` ϕ	distinct type parameters $\vec{\alpha}$
`type` $x[\vec{\alpha}] = \psi$	distinct type parameters $\vec{\alpha}$
`def` $f[\vec{\alpha}](\vec{x} : \vec{\psi})$ `->` ψ`:` s	distinct type parameters $\vec{\alpha}$; distinct parameters $\vec{x}$
`lambda` $\vec{x}$`:` e	distinct parameters $\vec{x}$
$\vec{u}, \vec{x}{=}\vec{u}'$	distinct keyword arguments $\vec{x}$
$\{\vec{w} : \vec{p}\}$	distinct keys $\vec{w}$
`case` p`:` s	distinct variables bound by pattern p
top-level statement t	`return` only within function bodies

Figure 2.3–4: Conditions ensuring every program is syntactically valid Python

2.3.2 Operators

The metavariable $\odot$ ranges over the supported binary operators other than `and` and `or`, and $\ominus$ over the supported unary operators, both enumerated in Figure 2.3–5. The operators `and` and `or` are distinct expression forms, since they need not evaluate both operands. The grammar of Figure 2.3–2 is intentionally ambiguous on operator forms; disambiguation is by the precedence and associativity given in Figure 2.3–5, which follow Python. It is ambiguous in the same way on tuples and tuple patterns, which are written with grouping parentheses where Python's grammar requires them. The empty tuple is written `()`, one-element tuples carry a trailing comma, `(1,)`, and trailing commas are permitted wherever Python's grammar permits them.

2.4 Modules and programs

Definition 2.6 (Program)**.** A PurePy *program* is a finite map $\mathcal{M}$ from *fully-qualified module names* (Figure 2.3–1) to *module bodies*, with a distinguished *main module* named `__main__`. We identify a module with its fully-qualified name, writing "module q" for the module named q. A module body is a sequence of imports $\vec{\imath}$, its

Level	Operators	Arity	Associativity
(loosest) 1	`or`	binary	left
2	`and`	binary	left
3	`not`	unary	—
4	`==` `!=` `<` `<=` `>` `>=` `in` `not in`	binary	left
5	`|`	binary	left
6	`+` `-`	binary	left
7	`*` `/` `//` `%`	binary	left
8	`+` `-`	unary	—
(tightest) 9	`**`	binary	right

Figure 2.3–5: Operator precedence

import prefix, each a *plain import* `import` q or a *from-import* `from` q `import` $\vec{x}$, followed by a top-level statement t; since imports are not statements, they can appear neither within a statement nor after one. After its imports, the body begins with the declaration `__name__: str` = name(q), where name(q) is the string literal naming q, binding the name of the module when the module loads (§3.5).

A second map $\mathcal{P}$ gives the predefined modules of Figure 2.4–2, which have no body: $\mathcal{P}$ takes each such name to its signature (Definition 2.7). The two domains are disjoint, and their union is prefix-closed: the parent of any module in $\mathrm{dom}(\mathcal{M}) \cup \mathrm{dom}(\mathcal{P})$ is itself in $\mathrm{dom}(\mathcal{M}) \cup \mathrm{dom}(\mathcal{P})$.[3] The maps $\mathcal{M}$ and $\mathcal{P}$ are fixed and ambient in all definitions that follow. A third map $\mathcal{E}(\mathcal{A})$ takes each predefined module to its environment for a run with command-line arguments $\mathcal{A}$ (Definition 2.8). Following `sys.argv` in Python, $\mathcal{A}$ is a sequence of strings beginning with the file name of the main module.

Definition 2.7 (Predefined signatures)**.** Define $\mathcal{P}$ to be the map taking each predefined module q to the signature which binds `__name__` at `str` and each member of q at the context entry given in Figure 2.4–1.

Definition 2.8 (Predefined environments for a run)**.** For a run with command-line arguments $\mathcal{A}$, define $\mathcal{E}(\mathcal{A})$ to be the map taking each predefined module q to the environment which binds `__name__` to name(q) and each member to the value given in Figure 2.4–2.

The predefined signatures name types that are themselves members, sometimes of another predefined module, so each is well-formed in their union. The class table of the predefined modules is

$$\Sigma_{\mathcal{P}} = \{\texttt{builtins.range} : \mathsf{ClassDef}_{\Gamma}(\texttt{builtins}, \chi, 1)\}$$

where χ defines the dataclass `range` with fields `start: int` and `stop: int` and Γ is the signature of `builtins`.

Definition 2.9 (Module containment)**.** Write $q \preceq q'$ (q *contains* q') when $q' = q$ or $q' = q.q^{\dagger}$ for some $q^{\dagger}$, and $\prec$ for strict containment.

The metafunctions on modules and classes are defined in §A.3.

[3] A source tree encodes such an $\mathcal{M}$: its modules are the `.py` files and the directories, each named by its path, so the domain is prefix-closed by construction. The statement of a module is the statement sequence of its `.py` file, or of its `__init__.py` for a directory, preceded by the `__name__` assignment; a directory without an `__init__.py` (a namespace package, PEP 420, https://peps.python.org/pep-0420/) contributes that assignment alone.

Module	Member	Entry
builtins	len	Callable[[Sized], int]
	range	Type(builtins.range)
	round	Callable[[float], int]
	None	Predef
	object	Predef
	bool	Predef
	int	Predef
	float	Predef
	str	Predef
	list	Predef
	dict	Predef
	tuple	Predef
sys	argv	list[str]
	exit	Callable[[int], Never]
typing	Callable	Predef
	Literal	Predef
	Never	Predef
	Sized	Predef
dataclasses	dataclass	Predef

Figure 2.4–1: Predefined signatures $\mathcal{P}$

Module	Member	Value	
`builtins`	`len`	Prim(f) where	
		$f(v) = \mathsf{val}(n)$	if $\mathrm{iter}(v) = v_1 \cdot \ldots \cdot v_n$
		$f(\vec{v}) = \mathsf{aborts}(\mathsf{TypeError})$	otherwise
	`range`	Type(`builtins.range`)	
	`round`	Prim(f) where	
		$f(n) = \mathsf{val}(n')$	if n' nearest integer to n, ties to even
		$f(n) = \mathsf{aborts}(\mathsf{OverflowError})$	if n infinite
		$f(n) = \mathsf{aborts}(\mathsf{ValueError})$	if $n = \mathsf{nan}$
		$f(v) = \mathsf{aborts}(\mathsf{excluded})$	if v Boolean
		$f(v, v') = \mathsf{aborts}(\mathsf{excluded})$	if v number or Boolean, and v' integer, Boolean or `None`
		$f(\vec{v}) = \mathsf{aborts}(\mathsf{TypeError})$	otherwise
	`None`	Predef(`builtins.None`)	
	`object`	Predef(`builtins.object`)	
	`bool`	Predef(`builtins.bool`)	
	`int`	Predef(`builtins.int`)	
	`float`	Predef(`builtins.float`)	
	`str`	Predef(`builtins.str`)	
	`list`	Predef(`builtins.list`)	
	`dict`	Predef(`builtins.dict`)	
	`tuple`	Predef(`builtins.tuple`)	
`sys`	`argv`	$[w_1, \ldots, w_n]$ where $\mathcal{A} = w_1 \cdot \cdots \cdot w_n$	
	`exit`	Prim(f) where	
		$f(n) = \mathsf{aborts}(\mathsf{SystemExit}(n))$	if n integer
		$f() = \mathsf{aborts}(\mathsf{excluded})$	
		$f(v) = \mathsf{aborts}(\mathsf{excluded})$	if v not an integer
		$f(\vec{v}) = \mathsf{aborts}(\mathsf{TypeError})$	otherwise
`typing`	`Callable`	Predef(`typing.Callable`)	
	`Literal`	Predef(`typing.Literal`)	
	`Never`	Predef(`typing.Never`)	
	`Sized`	Predef(`typing.Sized`)	
`dataclasses`	`dataclass`	Predef(`dataclasses.dataclass`)	

Figure 2.4–2: Predefined environments $\mathcal{E}(\mathcal{A})$

3 Well-formedness

PurePy excludes some Python behaviours statically and others only at runtime. Well-formedness is the static part, a judgement on programs defined in this chapter; the operational semantics (Chapter 4) is defined for well-formed programs. A program is well-formed when it is well typed, so the rules of this chapter give each expression a type, each statement a static outcome recording either the assignments made or the fact that the statement definitely returns, and each module a signature. The chapter introduces types and subtyping, then follows the syntax of Chapter 2, taking statements, expressions and patterns, then modules and programs, and Chapter 4 gives each of these a dynamic counterpart.

The rules for variables serve two aims: that a program has no mutable state, so a variable never changes value, and that a well-typed program is also well typed under mypy, unless mypy rejects the program because of a behaviour reported as a bug (§B.1.5). Within these limits the rules permit as much as possible. For the first aim, a variable is introduced by a declaration, parameter, import, pattern or generator, and is assigned at most once, either in its declaration or afterwards; an assignment never introduces a variable nor reassigns one. For the second, a scope is the body of a module, function, lambda or comprehension, as in Python, and the builtins enclose every module; a scope declares a name at most once and may shadow a name of an enclosing scope, hiding it throughout the scope.

Four syntactic analyses support the well-formedness rules: assigns(t), the set of variables declared or assigned anywhere in t (without descending into nested function definitions, which have their own scope, so that assigns(t) = dom(Δ) when $t : \mathsf{Assigns}(\Delta)$); declares($t$), the sequence of names declared in t, with duplicates, so that a scope can require its declarations to be distinct; binds(p), the set of variables introduced by a pattern p, and likewise binds($\vec{g}$) by the generators of qualifiers and binds(s) by the patterns within a statement; and captures(e), the set of variables from the enclosing scope that are captured by lambdas within expression e, and likewise captures($\vec{g}$) within qualifiers. All are defined in §A.1.

3.1 Types

$\Gamma \vdash q : \theta$

simple-var
$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

simple-module
$$\frac{x : \mathsf{Mod}(q, \Gamma') \in \Gamma}{\Gamma \vdash x : \mathsf{Mod}(q, \Gamma')}$$

simple-class
$$\frac{c : \mathsf{Type}(C) \in \Gamma}{\Gamma \vdash c : \mathsf{Type}(C)}$$

simple-alias
$$\frac{x : \mathsf{TypeAlias}_{\Gamma'}(q, \vec{\chi}, i) \in \Gamma}{\Gamma \vdash x : \mathsf{TypeAlias}_{\Gamma'}(q, \vec{\chi}, i)}$$

simple-scheme
$$\frac{x : \forall \vec{\alpha}.\, \tau \in \Gamma \qquad |\vec{\alpha}| > 0}{\Gamma \vdash x : \forall \vec{\alpha}.\, \tau}$$

simple-predef
$$\frac{x : \mathsf{Predef} \in \Gamma}{\Gamma \vdash x : \mathsf{Predef}}$$

qualified
$$\frac{\Gamma \vdash q : \mathsf{Mod}(q', \Gamma') \qquad \Gamma' \vdash x : \theta}{\Gamma \vdash q.x : \theta}$$

Figure 3.1–1: Name q resolves to context entry θ under Γ (`resolve_name`)

Function signatures, assignments and dataclass fields are annotated with type expressions, and a type expression denotes a type under a context and class table (Figure 3.1–2). Every definition carries an annotation for its parameters and return type, so the type of a definition is given rather than inferred from its body. Within a type expression, a class name applied to as many type arguments as the class has parameters resolves to an instantiated class, an alias name applied to type arguments resolves to the instantiation of its alias, type parameters in scope denote the type variables of those names, and all other names must be predefined names

$\Sigma; \Gamma \vdash \psi \rhd \tau$

ty-name
$$\frac{\nu : \mathsf{Predef} \in \Gamma}{\Sigma; \Gamma \vdash \nu \rhd \nu}$$

ty-var
$$\frac{\alpha : \mathsf{TypeVar} \in \Gamma}{\Sigma; \Gamma \vdash \alpha \rhd \alpha}$$

ty-class
$$\frac{\Gamma \vdash q : \mathsf{Type}(C) \qquad \Sigma; \Gamma \vdash \vec{\psi} \rhd \vec{\tau} \qquad |\vec{\tau}| = |\text{type-params}_\Sigma(C)|}{\Sigma; \Gamma \vdash q[\vec{\psi}] \rhd C[\vec{\tau}]}$$

ty-alias
$$\frac{\Gamma \vdash q : \mathsf{TypeAlias}_{\Gamma'}(q', \vec{\chi}, i) \qquad \chi_i = \texttt{type}\ x[\vec{\alpha}] = \psi' \qquad \Sigma; \text{type-context}_{\Gamma'}(q', \vec{\chi}, i) \vdash \psi' \rhd \tau \qquad \Sigma; \Gamma \vdash \vec{\psi} \rhd \vec{\sigma}}{\Sigma; \Gamma \vdash q[\vec{\psi}] \rhd [\vec{\sigma}/\vec{\alpha}]\tau}$$

ty-literal
$$\frac{\texttt{Literal} : \mathsf{Predef} \in \Gamma}{\Sigma; \Gamma \vdash \texttt{Literal}[\ell] \rhd \texttt{Literal}[\ell]}$$

ty-list
$$\frac{\texttt{list} : \mathsf{Predef} \in \Gamma \qquad \Sigma; \Gamma \vdash \psi \rhd \tau}{\Sigma; \Gamma \vdash \texttt{list}[\psi] \rhd \texttt{list}[\tau]}$$

ty-dict
$$\frac{\texttt{dict} : \mathsf{Predef} \in \Gamma \qquad \texttt{str} : \mathsf{Predef} \in \Gamma \qquad \Sigma; \Gamma \vdash \psi \rhd \tau}{\Sigma; \Gamma \vdash \texttt{dict}[\texttt{str}, \psi] \rhd \texttt{dict}[\texttt{str}, \tau]}$$

ty-tuple
$$\frac{\texttt{tuple} : \mathsf{Predef} \in \Gamma \qquad \Sigma; \Gamma \vdash \psi_i \rhd \tau_i \quad (\forall i)}{\Sigma; \Gamma \vdash \texttt{tuple}[\vec{\psi}] \rhd \texttt{tuple}[\vec{\tau}]}$$

ty-callable
$$\frac{\texttt{Callable} : \mathsf{Predef} \in \Gamma \qquad \Sigma; \Gamma \vdash \psi_i \rhd \sigma_i \quad (\forall i) \qquad \Sigma; \Gamma \vdash \psi \rhd \tau}{\Sigma; \Gamma \vdash \texttt{Callable}[[\vec{\psi}], \psi] \rhd \texttt{Callable}[[\vec{\sigma}], \tau]}$$

ty-union
$$\frac{\Sigma; \Gamma \vdash \psi \rhd \sigma \qquad \Sigma; \Gamma \vdash \psi' \rhd \tau}{\Sigma; \Gamma \vdash \psi \mid \psi' \rhd \sigma \mid \tau}$$

Figure 3.1–2: Type expression ψ denotes type τ under Γ and Σ (`resolve_type`)

(Figure 3.1–2). The function base-type(τ) discards the precision of a literal type, taking `Literal[42]` to `int` and otherwise acting as the identity (§A.2).

Lemmas about subtyping assume well-formed types and a well-formed class table (Figures 3.1–3 and 3.1–4). The base-class relation is acyclic by the definition of a class table rather than by well-formedness, since the auxiliary functions on classes are defined by recursion on that relation.

3.1.1 Subtyping

Subtyping (Figure 3.1–5) has every type below itself, a literal type below its base type, and an instantiated class $C[\vec{\tau}]$ below every type that its base class is below, once $\vec{\tau}$ is substituted into the arguments of the base (subty-class-extend), and below $C[\vec{\sigma}]$ when $\vec{\tau}$ and $\vec{\sigma}$ are equivalent (subty-class), so that type arguments are invariant, as for lists and dictionaries; a type variable is below itself and `object` alone, with `object` above every type and `Never` below every type. A type that is not a union has one *disjunct*, the type itself; the disjuncts of a union $\sigma \mid \tau$ are the disjuncts of σ together with the disjuncts of τ. A union is above each of its disjuncts and below every type that is above all of them. A tuple type is covariant, below another of the same length when each of its components is below the corresponding component. Two types are *equivalent*, $\sigma \equiv \tau$, when each is a subtype of the other; the notations $<:$ and $\equiv$ follow Pierce [2002]. Lists and dictionaries are related only when their components are equivalent, since both are invariant in mypy (§B.1.5). A callable type is contravariant in its parameters and covariant in its result, following mypy. The types with a length, namely lists, dictionaries, strings, tuples and `builtins.range[]`, are below `Sized`. An `int` is a `float`, as PEP 484 requires, so `x: float = 0` is well typed. A `bool` is not an `int`, although Python treats it as one.

Lemma 3.1 (Transitivity of subtyping; `subty_trans`). *If* $\vdash \Sigma\ \mathsf{wf}$, $\Sigma \vdash \sigma\ \mathsf{wf}$ *and* $\Sigma \vdash \tau\ \mathsf{wf}$, $\Sigma \vdash \sigma <: \tau$ *and* $\Sigma \vdash \tau <: \tau'$ *then* $\Sigma \vdash \sigma <: \tau'$.

Subtyping of well-formed types is therefore a preorder: reflexive and transitive but not antisymmetric, because distinct types such as $\sigma \mid \tau$ and $\tau \mid \sigma$ are equivalent.

Definition 3.1 (Comparability, join and meet). Two types are *comparable* when either is a subtype of the other. Two comparable types *join* to the larger type and two incomparable types to their union. Two comparable types

$\Sigma \vdash \tau$ wf

wf-ty-prim
$$\frac{}{\Sigma \vdash \nu \text{ wf}}$$

wf-ty-literal
$$\frac{}{\Sigma \vdash \texttt{Literal}[\ell] \text{ wf}}$$

wf-ty-var
$$\frac{}{\Sigma \vdash \alpha \text{ wf}}$$

wf-ty-list
$$\frac{\Sigma \vdash \tau \text{ wf}}{\Sigma \vdash \texttt{list}[\tau] \text{ wf}}$$

wf-ty-dict
$$\frac{\Sigma \vdash \tau \text{ wf}}{\Sigma \vdash \texttt{dict}[\texttt{str}, \tau] \text{ wf}}$$

wf-ty-tuple
$$\frac{\Sigma \vdash \tau_i \text{ wf} \quad (\forall i)}{\Sigma \vdash \texttt{tuple}[\vec{\tau}] \text{ wf}}$$

wf-ty-callable
$$\frac{\Sigma \vdash \sigma_i \text{ wf} \quad (\forall i) \qquad \Sigma \vdash \tau \text{ wf}}{\Sigma \vdash \texttt{Callable}[[\vec{\sigma}], \tau] \text{ wf}}$$

wf-ty-class
$$\frac{C \in \text{dom}(\Sigma) \qquad |\vec{\tau}| = |\text{type-params}_\Sigma(C)| \qquad \Sigma \vdash \tau_i \text{ wf} \quad (\forall i)}{\Sigma \vdash C[\vec{\tau}] \text{ wf}}$$

wf-ty-union
$$\frac{\Sigma \vdash \sigma \text{ wf} \qquad \Sigma \vdash \tau \text{ wf}}{\Sigma \vdash \sigma \mid \tau \text{ wf}}$$

(a) Type τ is well-formed (`wf_ty`)

$\Sigma \vdash \theta$ wf

wf-unbound
$$\frac{}{\Sigma \vdash \text{Unbound wf}}$$

wf-type-var
$$\frac{}{\Sigma \vdash \text{TypeVar wf}}$$

wf-mod
$$\frac{}{\Sigma \vdash \text{Mod}(q) \text{ wf}}$$

wf-predef
$$\frac{}{\Sigma \vdash \text{Predef wf}}$$

wf-du
$$\frac{\Sigma \vdash \tau \text{ wf}}{\Sigma \vdash \text{DU}(\tau) \text{ wf}}$$

wf-pu
$$\frac{\Sigma \vdash \theta \text{ wf} \qquad \theta \in \{\tau, \forall \vec{\alpha}.\, \tau\}}{\Sigma \vdash \text{PU}(\theta) \text{ wf}}$$

wf-scheme
$$\frac{\Sigma \vdash \texttt{Callable}[[\vec{\sigma}], \tau] \text{ wf}}{\Sigma \vdash \forall \vec{\alpha}.\, \texttt{Callable}[[\vec{\sigma}], \tau] \text{ wf}}$$

wf-class
$$\frac{C \in \text{dom}(\Sigma)}{\Sigma \vdash \text{Type}(C) \text{ wf}}$$

wf-mod-loaded
$$\frac{\Sigma \vdash \Gamma \text{ wf} \qquad \text{tyvars}(\Gamma) = \emptyset}{\Sigma \vdash \text{Mod}(q, \Gamma) \text{ wf}}$$

wf-type-alias
$$\frac{\Sigma \vdash \Gamma \text{ wf} \qquad \text{tyvars}(\Gamma) = \emptyset \qquad \chi_i = \texttt{type}\ x[\vec{\alpha}] = \psi \qquad \text{dom}(\text{class-table}_\Gamma(q, \vec{\chi})) \subseteq \text{dom}(\Sigma)}{\Sigma \vdash \text{TypeAlias}_\Gamma(q, \vec{\chi}, i) \text{ wf}}$$

$\Sigma \vdash \Gamma$ wf

wf-cxt
$$\frac{\Sigma \vdash \Gamma(x) \text{ wf} \quad (\forall x \in \text{dom}(\Gamma))}{\Sigma \vdash \Gamma \text{ wf}}$$

(b) Context entry θ and context Γ are well-formed (`wf_entry`, `wf_cxt`)

Figure 3.1–3: Well-formed types, context entries and contexts

meet to the smaller type; a meet of incomparable types distributes over a union, takes tuples componentwise and callables with parameters joined and results combined using meet, and is `Never` otherwise, since no value belongs to both types. So joining a type with itself gives that type, joining a class with its base class gives the base class, and joining `int` with `str` gives `int | str`. A sequence of types is joined by folding binary join over the sequence, with `Never` as the unit (§A.2).

Lemma 3.2 (Type lattice; `join_upper`). *Types ordered by subtyping in Σ, up to equivalence, form a bounded distributive lattice: $\sigma \vee_\Sigma \tau$ is the least upper bound of σ and τ, $\sigma \wedge_\Sigma \tau$ is their greatest lower bound, `object` is the greatest type and `Never` the least.*

$\Sigma \vdash C$ wf-class

wf-class-def

$$\frac{\Sigma(C) = \mathsf{ClassDef}_\Gamma(q, \vec{\chi}, i) \qquad \Sigma \vdash \Gamma \text{ wf} \qquad \text{tyvars}(\Gamma) = \emptyset \qquad \text{dom}(\text{class-table}_\Gamma(q, \vec{\chi})) \subseteq \text{dom}(\Sigma) \\ \Sigma \vdash D[\vec{\sigma}] \text{ wf} \quad D \neq \texttt{builtins.range} \quad \text{fv}(D[\vec{\sigma}]) \subseteq \vec{\alpha} \quad (\text{if base}_\Sigma(C) = D[\vec{\sigma}]) \\ \text{fields}_\Sigma(C) = \forall \vec{\alpha}.\, \vec{x} : \vec{\tau} \qquad \Sigma \vdash \tau_j \text{ wf} \quad \text{fv}(\tau_j) \subseteq \vec{\alpha} \quad (\forall j) \qquad \vec{x} \text{ distinct}}{\Sigma \vdash C \text{ wf-class}}$$

(a) Class C is well-formed

$\vdash \Sigma$ wf

wf-class-table

$$\frac{\Sigma \vdash C \text{ wf-class} \quad (\forall C \in \text{dom}(\Sigma))}{\vdash \Sigma \text{ wf}}$$

(b) Class table Σ is well-formed (`wf_class_table`)

Figure 3.1–4: Well-formed classes and class tables

$\Sigma \vdash \sigma <: \tau$

subty-refl

$$\frac{}{\Sigma \vdash \tau <: \tau}$$

subty-class-extend

$$\frac{\text{type-params}_\Sigma(C) = \vec{\alpha} \qquad \text{base}_\Sigma(C) = D[\vec{\sigma}] \qquad \Sigma \vdash D[[\vec{\tau}/\vec{\alpha}]\vec{\sigma}] <: \tau'}{\Sigma \vdash C[\vec{\tau}] <: \tau'}$$

subty-class

$$\frac{\Sigma \vdash \sigma_i \equiv \tau_i \quad (\forall i)}{\Sigma \vdash C[\vec{\sigma}] <: C[\vec{\tau}]}$$

subty-never

$$\frac{}{\Sigma \vdash \texttt{Never} <: \tau}$$

subty-object

$$\frac{}{\Sigma \vdash \tau <: \texttt{object}}$$

subty-int-float

$$\frac{}{\Sigma \vdash \texttt{int} <: \texttt{float}}$$

subty-sized

$$\frac{\tau \text{ a list, dictionary, string or tuple type, or } \texttt{builtins.range}[]}{\Sigma \vdash \tau <: \texttt{Sized}}$$

subty-list

$$\frac{\Sigma \vdash \sigma \equiv \tau}{\Sigma \vdash \texttt{list}[\sigma] <: \texttt{list}[\tau]}$$

subty-dict

$$\frac{\Sigma \vdash \sigma \equiv \tau}{\Sigma \vdash \texttt{dict}[\texttt{str}, \sigma] <: \texttt{dict}[\texttt{str}, \tau]}$$

subty-callable

$$\frac{\Sigma \vdash \tau_i <: \sigma_i \quad (\forall i) \qquad \Sigma \vdash \sigma <: \tau}{\Sigma \vdash \texttt{Callable}[[\sigma_1, \ldots, \sigma_n], \sigma] <: \texttt{Callable}[[\tau_1, \ldots, \tau_n], \tau]}$$

subty-tuple

$$\frac{\Sigma \vdash \sigma_i <: \tau_i \quad (\forall i)}{\Sigma \vdash \texttt{tuple}[\sigma_1, \ldots, \sigma_n] <: \texttt{tuple}[\tau_1, \ldots, \tau_n]}$$

subty-literal

$$\frac{\Sigma \vdash \text{base-type}(\ell) <: \tau}{\Sigma \vdash \texttt{Literal}[\ell] <: \tau}$$

subty-union-left

$$\frac{\Sigma \vdash \sigma <: \tau}{\Sigma \vdash \sigma <: \tau \mid \tau'}$$

subty-union-right

$$\frac{\Sigma \vdash \sigma <: \tau'}{\Sigma \vdash \sigma <: \tau \mid \tau'}$$

subty-union

$$\frac{\Sigma \vdash \tau <: \sigma \qquad \Sigma \vdash \tau' <: \sigma}{\Sigma \vdash \tau \mid \tau' <: \sigma}$$

Figure 3.1–5: Subtyping (`subty`)

3.2 Expressions

Within an expression the typing judgement gives each subexpression a type, and is bidirectional. *Synthesis* $\Sigma;\Gamma \vdash e \triangleright \tau$ reads a type out of an expression (Figures 3.2–1, 3.2–2 and 3.2–3), and *checking* $\Sigma;\Gamma \vdash e \triangleleft \tau$ tests an expression against a type supplied by its surroundings (Figure 3.2–5). The division follows the usual recipe for bidirectional systems [Davies and Pfenning, 2000, Dunfield and Krishnaswami, 2021]: introduction forms check, since a lambda's parameters cannot be annotated and an empty list does not determine its element type, and elimination forms synthesise, since a call, a subscript or an attribute reference starts from a subterm whose type is known. Python has no annotation form $(e : \tau)$, by which such systems usually let a checking form occur where synthesis is required, so PurePy gives each introduction form a synthesis rule wherever its parts determine the type, an approach also taken by Swift, Kotlin and Scala [Apple Inc., 2025, JetBrains, 2025, Odersky et al., 2025].

Variables, literals, attribute references, calls, constructor calls, operators and subscripts synthesise. A variable synthesises a type only when its context entry is a type (var), so a variable that is not definitely assigned, or a name bound at a type scheme, does not synthesise a type. A literal ℓ synthesises the literal type `Literal`$[\ell]$, except for a float literal, which synthesises `float`.

The fields of a class C form a field scheme $\text{fields}_\Sigma(C)$ over its type parameters, with one binding $x : \tau$ per field: the fields of its base class first, recursively and instantiated at the type arguments of the base, then the fields declared by C itself, each at its declared type; the fields of an instantiated class $C[\vec{\tau}]$ are the instantiation $\text{fields}_\Sigma(C)[\vec{\tau}]$. The order matters, because a constructor call pairs positional arguments with fields by position. A call $q(a)$ is a constructor call if q denotes a class. A constructor call pairs each argument with a field using field-map_Σ and infers type arguments $\vec{\tau}'$ for the class C that q resolves to, using type-args_Σ (§A.2) over the arguments that synthesise, as in callee-scheme (constr). The arguments are checked against the field types instantiated at $\vec{\tau}'$, and the call synthesises $C[\vec{\tau}']$. A constructor call of the form $q[\vec{\psi}](a)$ takes the type arguments denoted by $\vec{\psi}$ instead (constr-explicit). An attribute reference $e.x$ has the type of the field x of the class synthesised by e (attr-object). The signature of a predefined module binds `__name__` at `str` and each member at the type given in Figure 2.4–1 (Definition 2.7). A module attribute reference $e.x$ resolves as a qualified name to a definitely assigned member (attr-module). A variable bound to a module or class is not an expression on its own, and occurs only as the head of a qualified name. In a subscript $e[e']$ there is a rule for each container that e can synthesise: a list (subscript-list) or string (subscript-str) with e' of type `int`, and a dictionary with e' of type `str` (subscript-dict). Tuples have three rules: an index of type `int` gives the join of the component types (subscript-tuple-int), unless it synthesises a literal type, in which case it gives the component type at that position, counting from the end when the index is negative (subscript-tuple, subscript-tuple-neg). A subscript, attribute reference or call with a subject synthesising a union is typed at each disjunct, with a variable in place of the subject, and synthesises the join of the results (subscript-union, attr-union, call-union). Generators need no such rule, because elem-type_Σ takes a union to the join of its disjuncts' element types. Dunfield and Pfenning [2004] state union elimination once, for an arbitrary evaluation context; here union elimination is stated separately for each elimination form.

The call rule for a call $e(\vec{e})$ uses the auxiliary judgement $\Sigma;\Gamma \vdash e, \vec{e} \triangleright \vec{\sigma}, \sigma$ (Figure 3.2–4), which determines parameter types $\vec{\sigma}$ and return type σ from the callee along with its arguments. The remaining premises check the arguments against $\vec{\sigma}$, and the call synthesises σ. The auxiliary judgement has two rules. A callee that synthesises a `Callable` type gives the parameter and return types of that type (callee). A name bound at a type scheme $\forall\vec{\alpha}.\,\texttt{Callable}[[\vec{\sigma}], \sigma]$ instantiates the type parameters from the arguments (callee-scheme). The type arguments are collected by type-args_Σ (§A.2) over the arguments that synthesise, by a one-way unification of each parameter type with the synthesised type, in which the type parameters $\vec{\alpha}$ are the unknowns: a type parameter takes the type found at the same position in the synthesised type, with one found at several positions taking the join of those types. Arguments that do not synthesise, such as lambdas, are left out. The parameter and return types are instantiated at the type arguments given by the context, which must bind every type parameter.

Lambdas, applied lambdas, lists, tuples, dictionaries, conditional expressions and comprehensions check (lambda, call-lambda, list, tuple, dict, cond, list-comp, dict-comp). A name bound at a type scheme $\forall\vec{\alpha}.\,\sigma$ checks against a type τ by instantiating $\vec{\alpha}$ at the type arguments collected by type-args_Σ from σ and τ, provided the instantiated σ is below τ (var-scheme), so a function with type parameters can be passed where a `Callable`

$\Sigma; \Gamma \vdash e \rhd \tau$

var
$$\frac{x : \tau \in \Gamma}{\Sigma; \Gamma \vdash x \rhd \tau}$$

literal
$$\frac{}{\Sigma; \Gamma \vdash \ell \rhd \texttt{Literal}[\ell]}$$

literal-float
$$\frac{n \text{ a float literal}}{\Sigma; \Gamma \vdash n \rhd \texttt{float}}$$

attr-module
$$\frac{\Gamma \vdash e.x \ : \ \tau}{\Sigma; \Gamma \vdash e.x \rhd \tau}$$

attr-object
$$\frac{\Sigma; \Gamma \vdash e \rhd C[\vec{\tau}] \qquad x : \sigma \in \text{fields}_\Sigma(C)[\vec{\tau}]}{\Sigma; \Gamma \vdash e.x \rhd \sigma}$$

call
$$\frac{\Sigma; \Gamma \vdash e, \vec{e} \rhd \vec{\tau}, \tau \qquad \Sigma; \Gamma \vdash \vec{e} \lhd \vec{\tau} \qquad \tau \neq \texttt{None}}{\Sigma; \Gamma \vdash e(\vec{e}) \rhd \tau}$$

call-lambda-syn
$$\frac{\Sigma; \Gamma \vdash \vec{e} \rhd \vec{\tau} \qquad \Sigma; \Gamma \otimes \{\vec{x} : \vec{\tau}\} \vdash e \rhd \tau}{\Sigma; \Gamma \vdash (\texttt{lambda}\ \vec{x} : \ e)(\vec{e}) \rhd \tau}$$

constr
$$\frac{\begin{array}{c}\Gamma \vdash q \ : \ \mathsf{Type}(C) \qquad \text{fields}_\Sigma(C) = \forall \vec{\alpha}.\, \vec{x} : \vec{\sigma} \qquad \text{field-map}_\Sigma(C, a) = \{\vec{x} : \vec{e}\} \\ I = \{i \mid \Sigma; \Gamma \vdash e_i \rhd \tau_i\} \qquad \text{type-args}_\Sigma(\vec{\alpha}, (\sigma_i \mid i \in I), (\tau_i \mid i \in I))(\vec{\alpha}) = \vec{\tau}' \qquad \Sigma; \Gamma \vdash \vec{e} \lhd [\vec{\tau}'/\vec{\alpha}]\vec{\sigma}\end{array}}{\Sigma; \Gamma \vdash q(a) \rhd C[\vec{\tau}']}$$

constr-explicit
$$\frac{\Gamma \vdash q \ : \ \mathsf{Type}(C) \qquad \text{fields}_\Sigma(C) = \forall \vec{\alpha}.\, \vec{x} : \vec{\sigma} \qquad \Sigma; \Gamma \vdash \vec{\psi} \rhd \vec{\tau} \qquad \text{field-map}_\Sigma(C, a) = \{\vec{x} : \vec{e}\} \qquad \Sigma; \Gamma \vdash \vec{e} \lhd [\vec{\tau}/\vec{\alpha}]\vec{\sigma}}{\Sigma; \Gamma \vdash q[\vec{\psi}](a) \rhd C[\vec{\tau}]}$$

binop
$$\frac{\Sigma; \Gamma \vdash e \rhd \sigma \qquad \Sigma; \Gamma \vdash e' \rhd \sigma' \qquad \text{resolve-op}_\Sigma(\odot, \sigma, \sigma') = \tau^\dagger}{\Sigma; \Gamma \vdash e \odot e' \rhd \tau^\dagger}$$

unop
$$\frac{\Sigma; \Gamma \vdash e \rhd \sigma \qquad \ominus e \neq \text{-}\, n \quad (\forall n) \qquad \text{resolve-op}_\Sigma(\ominus, \sigma) = \tau^\dagger}{\Sigma; \Gamma \vdash \ominus e \rhd \tau^\dagger}$$

cond-syn
$$\frac{\Sigma; \Gamma \vdash e \lhd \texttt{bool} \qquad I = \{i \mid \Sigma; \Gamma \vdash e_i \rhd \sigma_i\} \neq \emptyset \qquad \Sigma; \Gamma \vdash e_j \lhd \bigvee_\Sigma(\sigma_i \mid i \in I) \quad (\forall j \notin I)}{\Sigma; \Gamma \vdash e_1 \ \texttt{if}\ e\ \texttt{else}\ e_2 \rhd \bigvee_\Sigma(\sigma_i \mid i \in I)}$$

and
$$\frac{\Sigma; \Gamma \vdash e \lhd \texttt{bool} \qquad \Sigma; \Gamma \vdash e' \lhd \texttt{bool}}{\Sigma; \Gamma \vdash e \ \texttt{and}\ e' \rhd \texttt{bool}}$$

or
$$\frac{\Sigma; \Gamma \vdash e \lhd \texttt{bool} \qquad \Sigma; \Gamma \vdash e' \lhd \texttt{bool}}{\Sigma; \Gamma \vdash e \ \texttt{or}\ e' \rhd \texttt{bool}}$$

Figure 3.2–1: Expression synthesis: names, literals, attributes, calls and operators (`synth_expr`)

type is expected. An expression that synthesises σ checks against any supertype of σ (check-syn), which applies only when no other rule concludes $\Sigma; \Gamma \vdash e \lhd \tau$, so both judgements are syntax-directed and an expression has at most one derivation in each. The simple-scheme resolution rule (Figure 3.1–1) requires the type parameters to be non-empty, so callee-scheme and var-scheme do not apply to names bound at `Callable` types, although they would conclude the same as callee and check-syn.

The forms that synthesise are conditional expressions when a branch synthesises, at the join of the two types when both branches synthesise and otherwise at the synthesising branch's type, the other branch checking against it (cond-syn); tuples when every component synthesises (syn-tuple); comprehensions when the body synthesises (syn-list-comp, syn-dict-comp); and non-empty lists and dictionaries when at least one element synthesises, the remaining elements checking against the join of the base types of the synthesising elements (syn-list, syn-dict). A list, dictionary or comprehension takes the base type of the type synthesised by its elements or body, since `list` and `dict` are invariant and a literal element type would keep the container from matching

$\Sigma; \Gamma \vdash e \rhd \tau$ (continued)

subscript-list

$$\frac{\Sigma; \Gamma \vdash e \rhd \texttt{list}[\tau] \qquad \Sigma; \Gamma \vdash e' \lhd \texttt{int}}{\Sigma; \Gamma \vdash e[e'] \rhd \tau}$$

subscript-str

$$\frac{\Sigma; \Gamma \vdash e \rhd \sigma \qquad \text{base-type}(\sigma) = \texttt{str} \qquad \Sigma; \Gamma \vdash e' \lhd \texttt{int}}{\Sigma; \Gamma \vdash e[e'] \rhd \texttt{str}}$$

subscript-dict

$$\frac{\Sigma; \Gamma \vdash e \rhd \texttt{dict}[\texttt{str}, \tau] \qquad \Sigma; \Gamma \vdash e' \lhd \texttt{str}}{\Sigma; \Gamma \vdash e[e'] \rhd \tau}$$

subscript-tuple

$$\frac{\Sigma; \Gamma \vdash e \rhd \texttt{tuple}[\tau_1, \ldots, \tau_m] \qquad \Sigma; \Gamma \vdash e' \rhd \texttt{Literal}[n] \qquad 0 \le n < m}{\Sigma; \Gamma \vdash e[e'] \rhd \tau_{n+1}}$$

subscript-tuple-neg

$$\frac{\Sigma; \Gamma \vdash e \rhd \texttt{tuple}[\tau_1, \ldots, \tau_m] \qquad \Sigma; \Gamma \vdash e' \rhd \texttt{Literal}[n] \qquad -m \le n < 0}{\Sigma; \Gamma \vdash e[e'] \rhd \tau_{m+n+1}}$$

subscript-tuple-int

$$\frac{\Sigma; \Gamma \vdash e \rhd \texttt{tuple}[\vec{\tau}] \qquad \Sigma; \Gamma \vdash e' \rhd \texttt{int}}{\Sigma; \Gamma \vdash e[e'] \rhd \bigvee_\Sigma(\vec{\tau})}$$

subscript-union

$$\frac{\Sigma; \Gamma \vdash e \rhd \sigma \mid \sigma' \qquad \Sigma; \Gamma \otimes \{y : \sigma\} \vdash y[e'] \rhd \tau \qquad \Sigma; \Gamma \otimes \{y : \sigma'\} \vdash y[e'] \rhd \tau' \qquad y \notin \text{fv}(e')}{\Sigma; \Gamma \vdash e[e'] \rhd \tau \vee_\Sigma \tau'}$$

attr-union

$$\frac{\Sigma; \Gamma \vdash e \rhd \sigma \mid \sigma' \qquad \Sigma; \Gamma \otimes \{y : \sigma\} \vdash y.x \rhd \tau \qquad \Sigma; \Gamma \otimes \{y : \sigma'\} \vdash y.x \rhd \tau'}{\Sigma; \Gamma \vdash e.x \rhd \tau \vee_\Sigma \tau'}$$

call-union

$$\frac{\Sigma; \Gamma \vdash e \rhd \sigma \mid \sigma' \qquad \Sigma; \Gamma \otimes \{y : \sigma\} \vdash y(\vec{e}) \rhd \tau \qquad \Sigma; \Gamma \otimes \{y : \sigma'\} \vdash y(\vec{e}) \rhd \tau' \qquad y \notin \text{fv}(\vec{e})}{\Sigma; \Gamma \vdash e(\vec{e}) \rhd \tau \vee_\Sigma \tau'}$$

binop-union-left

$$\frac{\begin{array}{l}\Sigma; \Gamma \vdash e \rhd \sigma \mid \sigma' \qquad \Sigma; \Gamma \vdash e' \rhd \rho \qquad \text{resolve-op}_\Sigma(\odot, \sigma \mid \sigma', \rho) \text{ undefined} \\ \Sigma; \Gamma \otimes \{y : \sigma\} \vdash y \odot e' \rhd \tau \qquad \Sigma; \Gamma \otimes \{y : \sigma'\} \vdash y \odot e' \rhd \tau' \qquad y \notin \text{fv}(e')\end{array}}{\Sigma; \Gamma \vdash e \odot e' \rhd \tau \vee_\Sigma \tau'}$$

binop-union-right

$$\frac{\begin{array}{l}\Sigma; \Gamma \vdash e \rhd \rho \qquad \rho \text{ not a union} \qquad \Sigma; \Gamma \vdash e' \rhd \sigma \mid \sigma' \qquad \text{resolve-op}_\Sigma(\odot, \rho, \sigma \mid \sigma') \text{ undefined} \\ \Sigma; \Gamma \otimes \{y : \sigma\} \vdash e \odot y \rhd \tau \qquad \Sigma; \Gamma \otimes \{y : \sigma'\} \vdash e \odot y \rhd \tau' \qquad y \notin \text{fv}(e)\end{array}}{\Sigma; \Gamma \vdash e \odot e' \rhd \tau \vee_\Sigma \tau'}$$

Figure 3.2–2: Expression synthesis: subscripts, and elimination at a union (`synth_expr`)

any other element type. An empty list or dictionary and a lambda have no synthesis rule at all, so checking is the only way to type them. A lambda applied to arguments is the exception, since the arguments give the parameter types, and the call then synthesises the type of the body or checks the body against an expected type (call-lambda-syn, call-lambda). A non-empty list has both, and the checking rule is still needed when the expected element type is wider than the type synthesised by the elements, since `list` is invariant.

Typing a comprehension types its qualifiers, which bind the variables that can be used in the body (Figure 3.2–5). A generator's variable is bound at elem-type$_\Sigma$ of the type synthesised by the iterated expression (§A.2). A tuple is iterated at the join of the base types of its components. A lambda within a comprehension may not capture a generator variable, a side condition of list-comp, dict-comp and qual-generator. Python closures are late-binding and the comprehension rebinds the variable on each iteration, so the lambda would see the variable's final value;

$\Sigma; \Gamma \vdash e \rhd \tau$ (continued)

syn-list
$$\frac{I = \{i \mid \Sigma; \Gamma \vdash e_i \rhd \sigma_i\} \neq \emptyset \qquad \tau = \bigvee_\Sigma(\text{base-type}(\sigma_i) \mid i \in I) \qquad \Sigma; \Gamma \vdash e_j \lhd \tau \quad (\forall j \notin I)}{\Sigma; \Gamma \vdash [e_1, \ldots, e_{n+1}] \rhd \texttt{list}[\tau]}$$

syn-tuple
$$\frac{\Sigma; \Gamma \vdash e_i \rhd \tau_i \quad (\forall i)}{\Sigma; \Gamma \vdash (\vec{e}) \rhd \texttt{tuple}[\vec{\tau}]}$$

syn-dict
$$\frac{\Sigma; \Gamma \vdash e_i \lhd \texttt{str} \quad (\forall i) \qquad I = \{i \mid \Sigma; \Gamma \vdash e'_i \rhd \sigma_i\} \neq \emptyset \qquad \tau = \bigvee_\Sigma(\text{base-type}(\sigma_i) \mid i \in I) \qquad \Sigma; \Gamma \vdash e'_j \lhd \tau \quad (\forall j \notin I)}{\Sigma; \Gamma \vdash \{e_1 : e'_1, \ldots, e_{n+1} : e'_{n+1}\} \rhd \texttt{dict}[\texttt{str}, \tau]}$$

syn-list-comp
$$\frac{\Sigma; \Gamma \vdash \vec{g} : \Delta \qquad \Sigma; \Gamma \otimes \Delta \vdash e \rhd \tau \qquad \text{captures}(e) \cap \text{dom}(\Delta) = \emptyset}{\Sigma; \Gamma \vdash [\, e \ \vec{g}] \rhd \texttt{list}[\text{base-type}(\tau)]}$$

syn-dict-comp
$$\frac{\Sigma; \Gamma \vdash \vec{g} : \Delta \qquad \Sigma; \Gamma \otimes \Delta \vdash e \lhd \texttt{str} \qquad \Sigma; \Gamma \otimes \Delta \vdash e' \rhd \tau \qquad \text{captures}((e, e')) \cap \text{dom}(\Delta) = \emptyset}{\Sigma; \Gamma \vdash \{e : e' \ \vec{g}\} \rhd \texttt{dict}[\texttt{str}, \text{base-type}(\tau)]}$$

Figure 3.2–3: Expression synthesis: tuples, lists, dictionaries and comprehensions (`synth_expr`)

$\Sigma; \Gamma \vdash e, \vec{e} \rhd \vec{\sigma}, \sigma$

callee
$$\frac{\Sigma; \Gamma \vdash e \rhd \texttt{Callable}[[\vec{\sigma}], \sigma]}{\Sigma; \Gamma \vdash e, \vec{e} \rhd \vec{\sigma}, \sigma}$$

callee-scheme
$$\frac{\Gamma \vdash q : \forall \vec{\alpha}.\, \texttt{Callable}[[\vec{\sigma}], \sigma] \qquad I = \{i \mid \Sigma; \Gamma \vdash e_i \rhd \tau_i\} \qquad \text{type-args}_\Sigma(\vec{\alpha}, (\sigma_i \mid i \in I), (\tau_i \mid i \in I))(\vec{\alpha}) = \vec{\tau}'}{\Sigma; \Gamma \vdash q, \vec{e} \rhd [\vec{\tau}'/\vec{\alpha}]\vec{\sigma}, [\vec{\tau}'/\vec{\alpha}]\sigma}$$

Figure 3.2–4: Callee e at arguments $\vec{e}$ synthesises parameter types $\vec{\sigma}$ and return type σ (`callee`)

in PurePy a lambda is only able to see the value of the generator variable at each iteration.

3.2.1 Operators

Each operator is overloaded as specified in Figure 3.2–6. The operator `|` has no overloads, so an expression $e \mid e'$ does not synthesise a type; the operator occurs only within type expressions, where $\psi \mid \psi'$ denotes a union (ty-union).

Definition 3.2 (Resolved overload)**.** A *resolved overload* $((\tau_1, \tau_2), \tau)$ pairs operand *bounds* with a return type; the judgements of the figure relate operand types to the resolved overloads of an operator.

The binop and unop rules (Figure 3.2–1) take the return type of the most specific overload at the operand types, resolve-op$_\Sigma$ (§A.2), so `1 + 2` is an `int` and `1 + 2.0` is a `float`. Binary operators also have rules for elimination at a union type (Figure 3.2–2), which apply when overload resolution fails at an operand of union type: the application synthesises a type at each disjunct, left operand first, and the type of the application is the join of those types (binop-union-left, binop-union-right). With `x` of type `int | str`, the expression `x * 2` synthesises `int | str`. The rules follow the Python typing specification for overloaded calls [Python Typing Council, 2025]. The unary operators need no such rule, since the overloads in Figure 3.2–6 apply at all subtypes of `bool` or `float`, including unions of such subtypes.

The judgement $\Sigma \vdash \tau$ needs-eq ξ (Figure 3.2–7a) states that equality at τ needs equality at type variable ξ, or at a callable type when ξ is ε. The judgement $\Sigma \vdash C$ needs-eq ξ (Figure 3.2–7b) states the same of class C,

$\Sigma; \Gamma \vdash e \lhd \tau$

check-syn

$$\frac{\Sigma; \Gamma \vdash e \rhd \sigma \qquad \Sigma \vdash \sigma <: \tau \qquad \text{no other rule concludes } \Sigma; \Gamma \vdash e \lhd \tau}{\Sigma; \Gamma \vdash e \lhd \tau}$$

lambda

$$\frac{\Sigma; \Gamma \otimes \{\vec{x} : \vec{\tau}\} \vdash e \lhd \tau}{\Sigma; \Gamma \vdash \texttt{lambda}\ \vec{x} : e \lhd \texttt{Callable}[[\vec{\tau}], \tau]}$$

var-scheme

$$\frac{\Gamma \vdash q : \forall \vec{\alpha}.\, \sigma \qquad \text{type-args}_\Sigma(\vec{\alpha}, \sigma, \tau)(\vec{\alpha}) = \vec{\tau}' \qquad \Sigma \vdash [\vec{\tau}'/\vec{\alpha}]\sigma <: \tau}{\Sigma; \Gamma \vdash q \lhd \tau}$$

call-lambda

$$\frac{\Sigma; \Gamma \vdash \vec{e} \rhd \vec{\tau} \qquad \Sigma; \Gamma \otimes \{\vec{x} : \vec{\tau}\} \vdash e \lhd \tau}{\Sigma; \Gamma \vdash (\texttt{lambda}\ \vec{x} : e)(\vec{e}) \lhd \tau}$$

list

$$\frac{\Sigma; \Gamma \vdash e_i \lhd \tau \quad (\forall i)}{\Sigma; \Gamma \vdash [\vec{e}] \lhd \texttt{list}[\tau]}$$

tuple

$$\frac{\Sigma; \Gamma \vdash e_i \lhd \tau_i \quad (\forall i)}{\Sigma; \Gamma \vdash (e_1, \ldots, e_n) \lhd \texttt{tuple}[\tau_1, \ldots, \tau_n]}$$

dict

$$\frac{\Sigma; \Gamma \vdash e_i \lhd \texttt{str} \quad (\forall i) \qquad \Sigma; \Gamma \vdash e'_i \lhd \tau \quad (\forall i)}{\Sigma; \Gamma \vdash \{e_1 : e'_1, \ldots, e_n : e'_n\} \lhd \texttt{dict}[\texttt{str}, \tau]}$$

cond

$$\frac{\Sigma; \Gamma \vdash e \lhd \texttt{bool} \qquad \Sigma; \Gamma \vdash e_1 \lhd \tau \qquad \Sigma; \Gamma \vdash e_2 \lhd \tau}{\Sigma; \Gamma \vdash e_1\ \texttt{if}\ e\ \texttt{else}\ e_2 \lhd \tau}$$

list-comp

$$\frac{\Sigma; \Gamma \vdash \vec{g} : \Delta \qquad \Sigma; \Gamma \otimes \Delta \vdash e \lhd \tau \qquad \text{captures}(e) \cap \text{dom}(\Delta) = \emptyset}{\Sigma; \Gamma \vdash [\, e\ \vec{g}] \lhd \texttt{list}[\tau]}$$

dict-comp

$$\frac{\Sigma; \Gamma \vdash \vec{g} : \Delta \qquad \Sigma; \Gamma \otimes \Delta \vdash e \lhd \texttt{str} \qquad \Sigma; \Gamma \otimes \Delta \vdash e' \lhd \tau \qquad \text{captures}((e, e')) \cap \text{dom}(\Delta) = \emptyset}{\Sigma; \Gamma \vdash \{e : e'\ \vec{g}\} \lhd \texttt{dict}[\texttt{str}, \tau]}$$

(a) Expression e checks against type τ

$\Sigma; \Gamma \vdash \vec{g} : \Delta$

quals-nil

$$\frac{}{\Sigma; \Gamma \vdash \varepsilon : \emptyset}$$

qual-guard

$$\frac{\Sigma; \Gamma \vdash e \lhd \texttt{bool} \qquad \Sigma; \Gamma \vdash \vec{g} : \Delta}{\Sigma; \Gamma \vdash \texttt{if}\ e \cdot \vec{g} : \Delta}$$

qual-generator

$$\frac{\Sigma; \Gamma \vdash e \rhd \tau \qquad \text{elem-type}_\Sigma(\tau) = \sigma \qquad \Sigma; \Gamma \otimes \{x : \sigma\} \vdash \vec{g} : \Delta \qquad x \notin \text{captures}(\vec{g})}{\Sigma; \Gamma \vdash \texttt{for}\ x\ \texttt{in}\ e \cdot \vec{g} : \{x : \sigma\} \otimes \Delta}$$

(b) Comprehension qualifiers $\vec{g}$ are well-formed and bind Δ

Figure 3.2–5: Expression checking (`check_expr`)

where ξ is a type parameter of C or ε, by the field types of C. The overloads of `==` and `!=` (Figure 3.2–6) require operand types τ with $\Sigma \Vdash \tau$ needs-eq ξ for every ξ, the *equality types*: comparing two values of type τ never reaches a type variable, whose instantiation is unknown, or a callable type, whose values cannot be compared. For example, a class C with type parameter α and a single field of type $C[\alpha] \mid \texttt{None}$, which mentions C through a type alias, does not need equality at α or at a callable type, so $C[\sigma]$ is an equality type for every σ. Comparisons of functions are therefore rejected statically, and values of an equality type are compared by structural equality, which is defined except at the dynamically excluded comparisons (§B.1.3).

The final rule of each table closes the judgement under base-typing, letting `1 == 0` use the equality overload at `int`, since two distinct literal types are not comparable; `2 ** 3` is still an `int`, since the direct instance's bounds (`int`, `Literal[3]`) lie below the base instance's (`float`, `float`). For this operator the binop rule is more precise than mypy, whose overloads give `Any` whenever the exponent is not a literal, whereas the most specific

$\Sigma \vdash \odot(\sigma_1, \sigma_2) : ((\tau_1, \tau_2), \tau)$

$\Sigma \vdash \odot(\sigma_1, \sigma_2) : ((\sigma_1, \sigma_2), \texttt{bool})$	if $\odot \in \{\texttt{==}, \texttt{!=}\}$ and σ_1, σ_2 comparable and $\Sigma \nvdash \sigma_i$ needs-eq ξ $(\forall i, \xi)$
$\Sigma \vdash \odot(\sigma_1, \texttt{list}[\tau]) : ((\sigma_1, \texttt{list}[\tau]), \texttt{bool})$	if $\odot \in \{\texttt{in}, \texttt{not in}\}$ and σ_1, τ comparable
$\Sigma \vdash \odot(\sigma_1, \texttt{tuple}[\vec{\tau}]) : ((\sigma_1, \texttt{tuple}[\vec{\tau}]), \texttt{bool})$	if $\odot \in \{\texttt{in}, \texttt{not in}\}$ and $\sigma_1, \bigvee_\Sigma(\vec{\tau})$ comparable
$\Sigma \vdash \odot(\sigma_1, \sigma_2) : ((\texttt{str}, \texttt{str}), \texttt{bool})$	if $\odot \in \{\texttt{in}, \texttt{not in}\}$ and $\Sigma \vdash \sigma_1, \sigma_2 <: \texttt{str}$
$\Sigma \vdash \odot(\sigma_1, \texttt{dict}[\texttt{str}, \tau]) : ((\texttt{str}, \texttt{dict}[\texttt{str}, \tau]), \texttt{bool})$	if $\odot \in \{\texttt{in}, \texttt{not in}\}$ and $\Sigma \vdash \sigma_1 <: \texttt{str}$
$\Sigma \vdash \odot(\sigma_1, \sigma_2) : ((\texttt{float}, \texttt{float}), \texttt{bool})$	if $\odot \in \{\texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}\}$ and $\Sigma \vdash \sigma_1, \sigma_2 <: \texttt{float}$
$\Sigma \vdash \odot(\sigma_1, \sigma_2) : ((\texttt{str}, \texttt{str}), \texttt{bool})$	if $\odot \in \{\texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}\}$ and $\Sigma \vdash \sigma_1, \sigma_2 <: \texttt{str}$
$\Sigma \vdash \odot(\sigma_1, \sigma_2) : ((\texttt{int}, \texttt{int}), \texttt{int})$	if $\odot \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{//}, \texttt{\%}\}$ and $\Sigma \vdash \sigma_1, \sigma_2 <: \texttt{int}$
$\Sigma \vdash \odot(\sigma_1, \sigma_2) : ((\texttt{float}, \texttt{float}), \texttt{float})$	if $\odot \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{//}, \texttt{\%}, \texttt{/}\}$ and $\Sigma \vdash \sigma_1, \sigma_2 <: \texttt{float}$
$\Sigma \vdash \texttt{**}(\sigma_1, \sigma_2) : ((\texttt{float}, \texttt{int}), \texttt{float})$	if $\Sigma \vdash \sigma_1 <: \texttt{float}$ and $\Sigma \vdash \sigma_2 <: \texttt{int}$
$\Sigma \vdash \texttt{+}(\sigma_1, \sigma_2) : ((\texttt{str}, \texttt{str}), \texttt{str})$	if $\Sigma \vdash \sigma_1, \sigma_2 <: \texttt{str}$
$\Sigma \vdash \texttt{+}(\texttt{list}[\sigma], \texttt{list}[\tau]) : ((\texttt{list}[\sigma], \texttt{list}[\tau]), \texttt{list}[\sigma \vee_\Sigma \tau])$	
$\Sigma \vdash \texttt{+}(\texttt{tuple}[\vec{\sigma}], \texttt{tuple}[\vec{\tau}]) : ((\texttt{tuple}[\vec{\sigma}], \texttt{tuple}[\vec{\tau}]), \texttt{tuple}[\vec{\sigma} +\!\!+ \vec{\tau}])$	
$\Sigma \vdash \texttt{*}(\sigma_1, \sigma_2) : ((\texttt{str}, \texttt{int}), \texttt{str})$	if $\Sigma \vdash \sigma_1 <: \texttt{str}$ and $\Sigma \vdash \sigma_2 <: \texttt{int}$
$\Sigma \vdash \texttt{*}(\sigma_1, \sigma_2) : ((\texttt{int}, \texttt{str}), \texttt{str})$	if $\Sigma \vdash \sigma_1 <: \texttt{int}$ and $\Sigma \vdash \sigma_2 <: \texttt{str}$
$\Sigma \vdash \texttt{*}(\texttt{list}[\tau], \sigma_2) : ((\texttt{list}[\tau], \texttt{int}), \texttt{list}[\tau])$	if $\Sigma \vdash \sigma_2 <: \texttt{int}$
$\Sigma \vdash \texttt{*}(\sigma_1, \texttt{list}[\tau]) : ((\texttt{int}, \texttt{list}[\tau]), \texttt{list}[\tau])$	if $\Sigma \vdash \sigma_1 <: \texttt{int}$
$\Sigma \vdash \texttt{**}(\sigma_1, \texttt{Literal}[n]) : ((\texttt{int}, \texttt{Literal}[n]), \texttt{int})$	if $\Sigma \vdash \sigma_1 <: \texttt{int}$ and n a non-negative integer
$\Sigma \vdash \texttt{**}(\sigma_1, \texttt{Literal}[n]) : ((\texttt{int}, \texttt{Literal}[n]), \texttt{float})$	if $\Sigma \vdash \sigma_1 <: \texttt{int}$ and n a negative integer
$\Sigma \vdash \odot(\sigma_1, \sigma_2) : ((\tau_1, \tau_2), \tau)$	if σ_1 or σ_2 a literal type and $\Sigma \vdash \odot(\text{base-type}(\sigma_1), \text{base-type}(\sigma_2)) : ((\tau_1, \tau_2), \tau)$

$\Sigma \vdash \ominus(\sigma) : (\tau, \tau')$

$\Sigma \vdash \texttt{not}(\sigma) : (\texttt{bool}, \texttt{bool})$	if $\Sigma \vdash \sigma <: \texttt{bool}$
$\Sigma \vdash \ominus(\sigma) : (\texttt{int}, \texttt{int})$	if $\ominus \in \{\texttt{+}, \texttt{-}\}$ and $\Sigma \vdash \sigma <: \texttt{int}$
$\Sigma \vdash \ominus(\sigma) : (\texttt{float}, \texttt{float})$	if $\ominus \in \{\texttt{+}, \texttt{-}\}$ and $\Sigma \vdash \sigma <: \texttt{float}$
$\Sigma \vdash \ominus(\sigma) : (\tau, \tau')$	if σ a literal type and $\Sigma \vdash \ominus(\text{base-type}(\sigma)) : (\tau, \tau')$

Figure 3.2–6: Operator overloads (`binop_overloads`, `unop_overloads`)

overload resolves to `float`.

Lemma 3.3 (Most specific overload; `resolve_binop_overload`, `resolve_unop_overload`)**.** *If neither σ_1 nor σ_2 is a subtype of `Never` and some overload of $\odot$ applies at (σ_1, σ_2), then* resolve-op$_\Sigma(\odot, \sigma_1, \sigma_2)$ *is defined, and similarly for* resolve-op$_\Sigma(\ominus, \sigma)$.

3.3 Statements

Definition 3.3 (Static outcome; `static_outcome`)**.** A well-typed statement has a *static outcome* R, which is either Returns or Assigns(Δ). The outcome is Returns when the statement *definitely returns* or definitely reaches `assert False`. Otherwise it is Assigns(Δ), where Δ is a context with an entry for each variable declared or assigned by the statement (Definition 2.1), recording whether the variable is definitely assigned, definitely unassigned or possibly unassigned after the statement.

Context operations $\otimes$ and $\oplus_\Sigma$ lift to a right semiring over static outcomes R, with Assigns($\emptyset$) as left and right unit for $\otimes$ and Returns as unit for $\oplus_\Sigma$ and zero for $\otimes$.

$\Sigma \vdash \tau$ needs-eq ξ

needs-eq-var

$$\frac{}{\Sigma \vdash \alpha \text{ needs-eq } \alpha}$$

needs-eq-callable

$$\frac{}{\Sigma \vdash \texttt{Callable}[[\vec{\tau}], \tau] \text{ needs-eq } \varepsilon}$$

needs-eq-list

$$\frac{\Sigma \vdash \tau \text{ needs-eq } \xi}{\Sigma \vdash \texttt{list}[\tau] \text{ needs-eq } \xi}$$

needs-eq-dict

$$\frac{\Sigma \vdash \tau \text{ needs-eq } \xi}{\Sigma \vdash \texttt{dict}[\texttt{str}, \tau] \text{ needs-eq } \xi}$$

needs-eq-tuple

$$\frac{\Sigma \vdash \tau_i \text{ needs-eq } \xi}{\Sigma \vdash \texttt{tuple}[\vec{\tau}] \text{ needs-eq } \xi}$$

needs-eq-union-left

$$\frac{\Sigma \vdash \sigma \text{ needs-eq } \xi}{\Sigma \vdash \sigma \mid \tau \text{ needs-eq } \xi}$$

needs-eq-union-right

$$\frac{\Sigma \vdash \tau \text{ needs-eq } \xi}{\Sigma \vdash \sigma \mid \tau \text{ needs-eq } \xi}$$

needs-eq-instance

$$\frac{\text{type-params}_\Sigma(C) = \vec{\alpha} \qquad \Sigma \vdash C \text{ needs-eq } \alpha_i \qquad \Sigma \vdash \sigma_i \text{ needs-eq } \xi}{\Sigma \vdash C[\vec{\sigma}] \text{ needs-eq } \xi}$$

needs-eq-instance-callable

$$\frac{\Sigma \vdash C \text{ needs-eq } \varepsilon}{\Sigma \vdash C[\vec{\sigma}] \text{ needs-eq } \varepsilon}$$

(a) Equality at type τ needs equality at type variable ξ, or at a callable type if ξ is ε (`needs_eq`)

$\Sigma \vdash C$ needs-eq ξ

needs-eq-class

$$\frac{\text{fields}_\Sigma(C) = \forall \vec{\alpha}.\, \vec{x} \colon \vec{\tau} \qquad \Sigma \vdash \tau_i \text{ needs-eq } \xi}{\Sigma \vdash C \text{ needs-eq } \xi}$$

(b) Equality at class C needs equality at its type parameter ξ, or at a callable type if ξ is ε (`needs_eq_class`)

Figure 3.2–7: Types compared by structural equality

Definition 3.4 (Override and merge on static outcomes; `override_outcome`, `merge_outcome`)**.**

$$\begin{aligned} \mathsf{Assigns}(\Delta) \oplus_\Sigma \mathsf{Assigns}(\Delta') &= \mathsf{Assigns}(\Delta \oplus_\Sigma \Delta') \\ R \oplus_\Sigma \mathsf{Returns} = \mathsf{Returns} \oplus_\Sigma R &= R \end{aligned}$$

$$\begin{aligned} \mathsf{Assigns}(\Delta) \otimes \mathsf{Assigns}(\Delta') &= \mathsf{Assigns}(\Delta \otimes \Delta') \\ R \otimes \mathsf{Returns} = \mathsf{Returns} \otimes R &= \mathsf{Returns} \end{aligned}$$

The $\mathsf{Returns} \otimes R$ case never arises because statement typing excludes unreachable statements.

The statement judgement $\Sigma; \Gamma; \zeta \vdash s : R$ says that under Γ and optional return type ζ, statement s has static outcome R (Figure 3.3–1); every rule passes ζ down unchanged. A sequence $s\, s'$ (seq) types s, then types s' under Γ overridden by the context assigned by s. A conditional combines its branches with $\oplus_\Sigma$, merging in $\mathsf{Assigns}(\emptyset)$ when there is no `else` branch (if, if-else). A `return` checks its expression against ζ, which must be of the form τ, and a bare `return` requires τ to be equivalent to `None` (return, return-none). An expression statement is typed by one of two rules. If its expression synthesises a type, the type is discarded (expr-stmt). A call to a callable with result `None` does not synthesise a type, following mypy (§B.1.5), which rejects such a call wherever a value is required; such calls are typed instead by call-stmt, which just checks the arguments against the parameter types. Both rules give the static outcome $\mathsf{Assigns}(\emptyset)$. A call statement with a callee synthesising a union is typed once per disjunct (call-stmt-union), following call-union for a call expression. A print statement requires its argument to synthesise a type, and is typed only when the return type ζ is absent, which excludes print statements from function bodies (print).

A variable declaration $x \colon \psi$ introduces x at the type denoted by ψ, without assigning it (declare); an assignment $x = e$ requires x to be a declared but unassigned variable, and checks e against its declared type (assign); the form $x \colon \psi = e$ declares and assigns x at once (declare-assign). Names introduced by declarations (declare, declare-assign), function definitions (defs) or pattern variables (check-var) must be unbound in the context. Variables start unbound, since the rules for function definitions and modules check the body under the context $\text{scope}(\vec{y}, s)$ (§A.1). The rules are unable to reject two bindings of a name in different branches of

$\Sigma; \Gamma; \zeta \vdash s : R$

pass

$$\frac{}{\Sigma; \Gamma; \zeta \vdash \texttt{pass} : \mathsf{Assigns}(\varnothing)}$$

declare

$$\frac{\Gamma(x) = \mathsf{Unbound} \qquad \Sigma; \Gamma \vdash \psi \rhd \tau}{\Sigma; \Gamma; \zeta \vdash x\colon\ \psi : \mathsf{Assigns}(\{x : \mathsf{DU}(\tau)\})}$$

assign

$$\frac{\Gamma(x) = \mathsf{DU}(\tau) \qquad \Sigma; \Gamma \vdash e \lhd \tau}{\Sigma; \Gamma; \zeta \vdash x = e : \mathsf{Assigns}(\{x : \tau\})}$$

declare-assign

$$\frac{\Gamma(x) = \mathsf{Unbound} \qquad \Sigma; \Gamma \vdash \psi \rhd \tau \qquad \Sigma; \Gamma \vdash e \lhd \tau}{\Sigma; \Gamma; \zeta \vdash x\colon\ \psi = e : \mathsf{Assigns}(\{x : \tau\})}$$

expr-stmt

$$\frac{\Sigma; \Gamma \vdash e \rhd \sigma}{\Sigma; \Gamma; \zeta \vdash e : \mathsf{Assigns}(\varnothing)}$$

print

$$\frac{\Sigma; \Gamma \vdash e \rhd \sigma}{\Sigma; \Gamma; \varepsilon \vdash \texttt{print}(e) : \mathsf{Assigns}(\varnothing)}$$

call-stmt

$$\frac{\Sigma; \Gamma \vdash e, \vec{e} \rhd \vec{\tau}, \texttt{None} \qquad \Sigma; \Gamma \vdash \vec{e} \lhd \vec{\tau}}{\Sigma; \Gamma; \zeta \vdash e(\vec{e}) : \mathsf{Assigns}(\varnothing)}$$

call-stmt-union

$$\frac{\Sigma; \Gamma \vdash e \rhd \sigma \mid \sigma' \qquad \Sigma; \Gamma \otimes \{y : \sigma\}; \zeta \vdash y(\vec{e}) : \mathsf{Assigns}(\varnothing) \qquad \Sigma; \Gamma \otimes \{y : \sigma'\}; \zeta \vdash y(\vec{e}) : \mathsf{Assigns}(\varnothing) \qquad y \notin \mathrm{fv}(\vec{e})}{\Sigma; \Gamma; \zeta \vdash e(\vec{e}) : \mathsf{Assigns}(\varnothing)}$$

assert

$$\frac{\Sigma; \Gamma \vdash e \lhd \texttt{bool} \qquad e \neq \texttt{False}}{\Sigma; \Gamma; \zeta \vdash \texttt{assert}\ e : \mathsf{Assigns}(\varnothing)}$$

assert-msg

$$\frac{\Sigma; \Gamma \vdash e \lhd \texttt{bool} \qquad \Sigma; \Gamma \vdash e' \lhd \texttt{str} \qquad e \neq \texttt{False}}{\Sigma; \Gamma; \zeta \vdash \texttt{assert}\ e, e' : \mathsf{Assigns}(\varnothing)}$$

assert-false

$$\frac{}{\Sigma; \Gamma; \zeta \vdash \texttt{assert False} : \mathsf{Returns}}$$

assert-false-msg

$$\frac{\Sigma; \Gamma \vdash e' \lhd \texttt{str}}{\Sigma; \Gamma; \zeta \vdash \texttt{assert False}, e' : \mathsf{Returns}}$$

return-none

$$\frac{\Sigma \vdash \tau \equiv \texttt{None}}{\Sigma; \Gamma; \tau \vdash \texttt{return} : \mathsf{Returns}}$$

return

$$\frac{\Sigma; \Gamma \vdash e \lhd \tau}{\Sigma; \Gamma; \tau \vdash \texttt{return}\ e : \mathsf{Returns}}$$

if

$$\frac{\Sigma; \Gamma \vdash e_i \lhd \texttt{bool} \quad (\forall i) \qquad \Sigma; \Gamma; \zeta \vdash s_i : R_i \quad (\forall i)}{\Sigma; \Gamma; \zeta \vdash \texttt{if}\ e_1 :\ s_1\ \texttt{elif}\ e_2 :\ s_2\ \ldots\ \texttt{elif}\ e_{n+1} :\ s_{n+1} : R_1 \oplus_\Sigma \ldots \oplus_\Sigma R_{n+1} \oplus_\Sigma \mathsf{Assigns}(\varnothing)}$$

if-else

$$\frac{\Sigma; \Gamma \vdash e_i \lhd \texttt{bool} \quad (\forall i) \qquad \Sigma; \Gamma; \zeta \vdash s_i : R_i \quad (\forall i)}{\Sigma; \Gamma; \zeta \vdash \texttt{if}\ e_1 :\ s_1\ \texttt{elif}\ e_2 :\ s_2\ \ldots\ \texttt{elif}\ e_{n+1} :\ s_{n+1}\ \texttt{else} :\ s_{n+2} : R_1 \oplus_\Sigma \ldots \oplus_\Sigma R_{n+2}}$$

seq

$$\frac{\Sigma; \Gamma; \zeta \vdash s : \mathsf{Assigns}(\Delta) \qquad \Sigma; \Gamma \otimes \Delta; \zeta \vdash s' : R}{\Sigma; \Gamma; \zeta \vdash s\ s' : \mathsf{Assigns}(\Delta) \otimes R}$$

Figure 3.3–1: Given optional return type ζ, statement s has static outcome R (`check_stmt`)

a conditional or cases of a match, since no context holds both. The condition therefore falls to the function scope, which is undefined for scopes binding a name twice, by parameter, import, declaration or pattern, unless the name is bound by two patterns. Under mypy a redeclaration is an error (§B.1.5), so the condition keeps well-typed programs within those accepted by mypy. The exemption for patterns allows the cases of a match to capture the same name, which is also permitted by mypy.

The conditions of `if` and `assert` check against `bool`, and an assertion message against `str` (assert, assert-msg). An assertion of the literal `False` has outcome Returns, like a `return` (assert-false, assert-false-msg). A function body therefore has outcome Returns if each of its branches ends in a `return` or in `assert False`, and in both module and function bodies, a variable assigned in one branch of an `if` is definitely assigned after the `if`

$\Sigma; \Gamma; \zeta \vdash s : R$

defs

$$\frac{\Gamma(f) = \mathsf{Unbound} \quad (\forall f \in \mathrm{declares}(\vec{d})) \qquad \mathrm{declares}(\vec{d})\ \text{distinct} \qquad \Sigma; \Gamma \vdash d_i \rhd \pi_i \quad (\forall i) \qquad \Delta' = \{\mathrm{declares}(\vec{d}) : \vec{\pi}\} \qquad \Sigma; \Gamma \otimes \Delta' \vdash d_i \lhd \pi_i \quad (\forall i)}{\Sigma; \Gamma; \zeta \vdash \vec{d} : \mathsf{Assigns}(\Delta')}$$

match

$$\frac{\Sigma; \Gamma \vdash e \rhd \sigma \qquad \Sigma; \Gamma; \zeta \vdash \sigma, \mathrm{shapes}_\Sigma(\sigma, \emptyset), \vec{b} : S, \texttt{Never}, \emptyset}{\Sigma; \Gamma; \zeta \vdash \texttt{match}\ e\texttt{:}\ \vec{b} : S}$$

match-partial

$$\frac{\Sigma; \Gamma \vdash e \rhd \sigma \qquad \Sigma; \Gamma; \zeta \vdash \sigma, \mathrm{shapes}_\Sigma(\sigma, \emptyset), \vec{b} : S, \tau, L \qquad \tau \neq \texttt{Never}}{\Sigma; \Gamma; \zeta \vdash \texttt{match}\ e\texttt{:}\ \vec{b} : S \oplus_\Sigma \mathsf{Assigns}(\emptyset)}$$

Figure 3.3–2: Statement typing, continued (function definitions and match) (`check_stmt`)

$\Sigma; \Gamma \vdash d \rhd \pi$

def-signature

$$\frac{\{\vec{\alpha}\} \cap \mathrm{tyvars}(\Gamma) = \emptyset \qquad \Gamma' = \Gamma \otimes \{\alpha_1 : \mathsf{TypeVar}, \ldots, \alpha_n : \mathsf{TypeVar}\} \qquad \Sigma; \Gamma' \vdash \vec{\psi} \rhd \vec{\sigma} \qquad \Sigma; \Gamma' \vdash \psi \rhd \tau}{\Sigma; \Gamma \vdash \texttt{def}\ f[\vec{\alpha}](\vec{x}\texttt{:}\ \vec{\psi})\ \texttt{->}\ \psi\texttt{:}\ s \rhd \forall \vec{\alpha}.\, \texttt{Callable}[[\vec{\sigma}], \tau]}$$

(a) Definition d synthesises type scheme π (`def_signature`)

$\Sigma; \Gamma \vdash d \lhd \pi$

def-body

$$\frac{\Gamma' = \Gamma \otimes \{\alpha_1 : \mathsf{TypeVar}, \ldots, \alpha_n : \mathsf{TypeVar}\} \qquad \Sigma; \Gamma' \otimes \{\vec{x} : \vec{\sigma}\} \otimes \mathrm{scope}(\vec{x}, s); \tau \vdash s : R \qquad R = \mathsf{Returns}\ \text{or}\ \Sigma \vdash \tau \equiv \texttt{None}}{\Sigma; \Gamma \vdash \texttt{def}\ f[\vec{\alpha}](\vec{x}\texttt{:}\ \vec{\psi})\ \texttt{->}\ \psi\texttt{:}\ s \lhd \forall \vec{\alpha}.\, \texttt{Callable}[[\vec{\sigma}], \tau]}$$

(b) Definition d checks against type scheme π (`check_def`)

Figure 3.3–3: Function definitions

$\Sigma; \Gamma; \zeta \vdash \tau, L, \vec{b} : S, \tau', L'$

match-case

$$\frac{\Sigma; \Gamma \vdash p \lhd \tau \dashv \Delta \qquad \Sigma; \Gamma \vdash L, p \rightsquigarrow K, L' \qquad \Sigma; \Gamma \otimes \Delta; \zeta \vdash s : R}{\Sigma; \Gamma; \zeta \vdash \tau, L, \texttt{case}\ p\texttt{:}\ s : \mathsf{Assigns}(\Delta) \otimes R, \mathrm{remaining}_\Sigma(\Gamma, \tau, p), L'}$$

match-cases

$$\frac{\Sigma; \Gamma; \zeta \vdash \tau, L, b : S, \sigma, L' \qquad \Sigma; \Gamma; \zeta \vdash \sigma, L', \vec{b} : S', \tau', L^\dagger}{\Sigma; \Gamma; \zeta \vdash \tau, L, b \cdot \vec{b} : S \oplus_\Sigma S', \tau', L^\dagger}$$

Figure 3.3–4: Given optional return type ζ, cases $\vec{b}$ have static outcome S and transform remaining type τ and residual L to τ' and L' (`match_cases`)

$\Sigma; q; \Gamma \vdash t \ : \ R \dashv \Sigma'$

top-stmt
$$\frac{\Sigma; \Gamma; \varepsilon \vdash s \ : \ R}{\Sigma; q; \Gamma \vdash s \ : \ R \dashv \Sigma}$$

top-seq
$$\frac{\Sigma; q; \Gamma \vdash t \ : \ \mathsf{Assigns}(\Delta) \dashv \Sigma^{\dagger} \qquad \Sigma^{\dagger}; q; \Gamma \otimes \Delta \vdash t' \ : \ R \dashv \Sigma'}{\Sigma; q; \Gamma \vdash t \ t' \ : \ \mathsf{Assigns}(\Delta) \otimes R \dashv \Sigma'}$$

types
$$\frac{\Delta = \{\mathsf{declares}(\vec{\chi}) \colon \mathsf{type\text{-}entry}_{\Gamma}(q, \vec{\chi})\} \qquad \Sigma' = \Sigma \otimes \mathsf{class\text{-}table}_{\Gamma}(q, \vec{\chi}) \qquad \Sigma'; \mathsf{type\text{-}context}_{\Gamma}(q, \vec{\chi}, i) \vdash \chi_i \quad (\forall i)}{\Sigma; q; \Gamma \vdash \vec{\chi} \ : \ \mathsf{Assigns}(\Delta) \dashv \Sigma'}$$

(a) Top-level statement t of module q has static outcome R, extending Σ to Σ' (`check_top_stmt`)

$\Sigma; \Gamma \vdash \chi$

class
$$\frac{\texttt{dataclass} \colon \mathsf{Predef} \in \Gamma \qquad \text{own-fields}(\phi) = \vec{x} \colon \vec{\psi}, \ \vec{x} \text{ distinct} \qquad \Sigma; \Gamma \vdash \psi_i \rhd \sigma_i \quad (\forall i)}{\Sigma; \Gamma \vdash \texttt{@dataclass class } c[\vec{\alpha}] \ : \ \phi}$$

class-extend
$$\frac{\begin{array}{l}\Gamma \vdash q' \ : \ \mathsf{Type}(D) \qquad D \neq \texttt{builtins.range} \qquad \Sigma; \Gamma \vdash q'[\vec{\psi'}] \rhd D[\vec{\sigma'}] \qquad \texttt{dataclass} \colon \mathsf{Predef} \in \Gamma \\ \text{own-fields}(\phi) = \vec{x} \colon \vec{\psi}, \ \vec{x} \text{ distinct} \qquad \Sigma; \Gamma \vdash \psi_i \rhd \sigma_i \quad (\forall i) \qquad \mathsf{fields}_{\Sigma}(D) = \forall \vec{\alpha}'. \, \vec{y} \colon \vec{\tau} \qquad \{\vec{x}\} \cap \{\vec{y}\} = \emptyset\end{array}}{\Sigma; \Gamma \vdash \texttt{@dataclass class } c[\vec{\alpha}](q'[\vec{\psi'}]) \ : \ \phi}$$

type-alias
$$\frac{\Sigma; \Gamma \vdash \psi \rhd \tau}{\Sigma; \Gamma \vdash \texttt{type } x[\vec{\alpha}] = \psi}$$

(b) Type definition χ is well-formed (`check_type_def`)

Figure 3.3–5: Top-level statements and type definitions

when the other branch reaches `assert False`, and mypy accepts both. A module whose body definitely reaches `assert False` is ill-formed, because the top-seq and module rules require outcome Assigns(Δ). An assertion of any other expression has outcome Assigns($\emptyset$), even at type `Literal[False]`. Here mypy is more precise, and also treats the code after an assertion as unreachable when the asserted expression is built from `False` with `not`, `and` and `or`, or has the declared type `Literal[False]`. Giving outcome Returns to every assertion at type `Literal[False]` would accept more than mypy, however. The expression `(False,)[0]` synthesises `Literal[False]`, from the type of the first component, whereas mypy types it at `bool` (Figure B.1–4).

A `match` scrutinee synthesises a type σ, and its cases $\vec{b}$ are checked in turn by the auxiliary judgement $\Sigma; \Gamma; \zeta \vdash \tau, L, \vec{b} \ : \ S, \tau', L'$ (Figure 3.3–4), which threads a remaining type, starting from σ, and a residual L of shapes, starting from the shapes of σ (Figure 3.4–3), through the cases, merging their outcomes into S. The match rule applies when the final remaining type is `Never`, meaning the match is exhaustive; the match-partial rule applies otherwise and also merges in Assigns($\emptyset$), the outcome of the path on which the scrutinee is not matched by any of the cases. The remaining type, the residual, reachability and exhaustiveness are the subject of §3.4.

3.3.1 Function definitions

The defs rule for a mutual def region of function definitions synthesises a type scheme for each definition (Figure 3.3–3), binds all the function names of the mutual def region at the synthesised type schemes, and checks each definition against its type scheme. For a definition `def` $f[\vec{\alpha}](\vec{x} \colon \vec{\psi})$ `->` $\psi \colon \ s$, the auxiliary judgements $\Sigma; \Gamma \vdash d \rhd \pi$ and $\Sigma; \Gamma \vdash d \lhd \pi$ synthesise the type scheme from the annotations (def-signature) and check the body s (def-body). Both rules extend the context with TypeVar entries for the type parameters $\vec{\alpha}$, which

must not intersect with the type variables already in scope (tyvars(Γ), §A.1), which may occur in the types of existing entries; a second binding of the same name would capture those occurrences. A type scheme without type parameters is identical to its `Callable` type, so the defs rule binds the name of a function without type parameters as a variable of that type. The type schemes of functions with type parameters are instantiated at a call, by the argument types (callee-scheme), and when checked against a `Callable` type, by that type (var-scheme). Since annotations denote types, not type schemes, declarations and assignments never bind variables at type schemes, only at types.

In the body, the parameters $\vec{x}$ are bound at the types denoted by $\vec{\psi}$; variables declared or assigned in s start unbound, emulating Python's scoping of function bodies, which prefers a locally bound name to one visible in an enclosing scope. A variable of the enclosing scope referenced by the body must be definitely assigned at the point at which the function is defined, whereas Python resolves the reference only when the body runs. As with any scope, the names declared in the function, including the parameters $\vec{x}$, must be distinct from one another and from any pattern variables in s. The function names of the mutual def region are declared in the enclosing scope rather than in the function, are distinct and unbound there (defs), and must be distinct from the other names declared in that scope.

The body s is checked with the return type denoted by ψ, and must have static outcome Returns unless that type is equivalent to `None`, so a function that declares a result definitely returns one. A return type such as `int | None` does not exempt the body from returning, nor permit a bare `return`, for compatibility with mypy (§B.1.5).

3.3.2 Top-level statements

Top-level statements are checked by a separate judgement, $\Sigma; q; \Gamma \vdash t : R \dashv \Sigma'$ (Figure 3.3–5): a sequence composes in the same way as a statement sequence, a mutual type region of class definitions and `type` statements is a top-level form, checked by the types rule, and a plain statement is checked with no return type (top-stmt), a `return` outside a function being excluded syntactically (Figure 2.3–4).

Within a mutual type region, a `type` statement may name any class or alias of the type region, in any order, but a class definition may name only earlier classes and aliases, because Python 3.12 evaluates the field annotations and base class of a class when the class statement runs, but evaluates the body of a `type` statement only when the alias is first used.[1] Recursive datatypes are therefore declared with their aliases before their classes, as in `type Expr = Num | Add` followed by definitions of `Num` and of `Add` with fields of type `Expr`. A cyclic alias such as `type T = list[T]` has no derivation, because resolving its body resolves the alias again.

The types rule (Figure 3.3–5) binds the names of a mutual type region $\vec{\chi}$ and adds its classes to the class table before checking its definitions. The context entry bound to the name declared by χ_i is $\text{type-entry}_\Gamma(q, \vec{\chi}, i)$ (§A.1): the class $\mathsf{Type}(q.c)$ if χ_i is a class definition, and the alias $\mathsf{TypeAlias}_\Gamma(q, \vec{\chi}, i)$ if it is a `type` statement. The class table is extended with $\text{class-table}_\Gamma(q, \vec{\chi})$ (§A.1), a class table defining all the classes of the type region. A class thus has three static forms, as a function has: the class statement χ_i, which defines it; the context entry $\mathsf{Type}(q.c)$, by which a variable refers to it by name; and $\mathsf{ClassDef}_\Gamma(q, \vec{\chi}, i)$ in the class table, the class statement together with its type region and defining context, from which its fields and base class are read. For a function the three are the `def` statement, the type scheme bound to its name and the closure $\mathsf{Def}_\Gamma(\rho, \vec{d}, i)$. Aliases and class definitions under their defining contexts carry the type region and defining context Γ, because alias bodies, field annotations and base classes are resolved at each use rather than when the type region is checked. Each definition χ_i is checked under the extended class table and the context $\text{type-context}_\Gamma(q, \vec{\chi}, i)$ (§A.1), by the auxiliary judgement $\Sigma; \Gamma \vdash \chi$, which binds every name of the type region for a `type` statement and only the earlier names for a class definition, as above.

The judgement $\Sigma; \Gamma \vdash \chi$ checks a class definition by resolving the field annotations, which may mention the type parameters of the class, and requiring the fields it declares to be distinct and, when it extends another class, the base class to name a class rather than an alias, not to name `range`, which Python rejects as a base class, and to resolve to an instantiated class, and the additional fields not to repeat an inherited field (class, class-extend). Class definitions also require the predefined name `dataclass` (Figure 2.4–1) to be in scope; every PurePy class must declare itself a dataclass. A `type` statement `type` $x[\vec{\alpha}]$ `=` ψ, such as `type Pair[T] = tuple[T, T]`,

[1] PurePy does not support `from __future__ import annotations`, under which Python 3.12 defers evaluation of annotations.

declares x as an alias of the type denoted by ψ with the type parameters $\vec{\alpha}$ in scope (type-alias). An application of the alias such as `Pair[int]` denotes that type at the given arguments (ty-alias), here `tuple[int, int]`, so types never mention aliases.

3.4 Pattern matching

To check a `match` with scrutinee type σ, the auxiliary judgement $\Sigma;\Gamma;\zeta \vdash \tau, L, \vec{b} : S, \tau', L'$ (Figure 3.3–4) threads a *remaining type* τ and a *residual* L through the cases, both over-approximations of the values of σ which are not matched by any of the earlier cases. The residual is an ordered set of *shapes* (Figure 3.4–3), each denoting a set of values of a single type, and starts from the shapes of σ; the remaining type is coarser, a type rather than a set of shapes, and starts from σ itself. Two auxiliary functions narrow a type τ by a pattern p (§A.2): matched-as$_\Sigma(\Gamma, \tau, p)$ is the type to which τ is narrowed if p matches, and remaining$_\Sigma(\Gamma, \tau, p)$ is the type remaining if p fails. Each case's pattern is checked against the remaining type (§3.4.1), binding its pattern variables, and matched against the residual (§3.4.3). Reachability is decided by the residual: a case is *reachable* if its pattern matches at least one shape of the residual, and a match with an unreachable case is ill-formed. For a reachable case, the body is checked under the bindings and its outcome merged into S; the shapes left after matching and the type remaining if the pattern fails become the residual and remaining type for the case that follows.

A match is *exhaustive* when its cases between them match every value of σ. Exhaustiveness is decided by the final remaining type, which the match rule requires to be `Never`, with match-partial applying otherwise. The remaining type thus determines exhaustiveness as well as the types of pattern variables. The Python typing specification leaves narrowing largely unspecified,[2] so the remaining type emulates the narrowing of a match scrutinee observed in mypy (§B.1.5), which subtracts a class only for a constructor pattern that matches every value of the class, and a tuple only through one refutable component, i.e. one whose pattern can fail to match a value of the component's type. Cases that between them cover a field of a class therefore leave an empty residual but a remaining type other than `Never`, so a function which returns from each of the cases is rejected as not always returning, and at a class with a field of type `int | None` a case `Box(n)` after a case `Box(None)` binds `n` at `int | None` rather than at `int`. The usefulness algorithm for ML [Maranget, 2007] also checks reachability and exhaustiveness with a single relation. The residual plays the role of the term descriptions of Sestoft [1996], and the uncovered sets of Karachalias et al. [2015] likewise let the type exclude cases.

3.4.1 Pattern checking

A pattern must *check* against the remaining type, $\Sigma;\Gamma \vdash p \lhd \tau \dashv \Delta$ (Figure 3.4–2), before it is matched against the residual (match-case). The bindings Δ give the types of the pattern variables. The judgement holds when every tuple pattern in p is at a type which has no list or range values, every list pattern is at a type which has no tuple or range values, every dictionary pattern is at a dictionary type or at a type which has no dictionary values, every int literal pattern is at a type which has no float values, and every constructor pattern is at a type at which its class has an instance or with which its class is incomparable, as defined below. List, tuple, range, dictionary and float values are defined by subtyping:

$$\begin{aligned}
\tau \text{ has list values} &\iff \Sigma \vdash \texttt{list}[\sigma] <: \tau \text{ for some } \sigma, \text{ or a disjunct of } \tau \text{ is a type variable} \\
\tau \text{ has tuple values} &\iff \Sigma \vdash \texttt{tuple}[\vec{\sigma}] <: \tau \text{ for some } \vec{\sigma}, \text{ or a disjunct of } \tau \text{ is a type variable} \\
\tau \text{ has range values} &\iff \Sigma \vdash \texttt{builtins.range}[] <: \tau, \text{ or a disjunct of } \tau \text{ is a type variable} \\
\tau \text{ has dictionary values} &\iff \Sigma \vdash \texttt{dict}[\texttt{str}, \sigma] <: \tau \text{ for some } \sigma, \text{ or a disjunct of } \tau \text{ is a type variable} \\
\tau \text{ has float values} &\iff \Sigma \vdash \texttt{float} <: \tau, \text{ or a disjunct of } \tau \text{ is a type variable}
\end{aligned}$$

So list types, `Sized` and `object` have list values; tuple types, `Sized` and `object` have tuple values; `builtins.range[]`, `Sized` and `object` have range values; dictionary types, `Sized` and `object` have dictionary values; `float` and `object` have float values; and a type variable, which may stand for any type, has all five kinds of value. The rules check a sub-pattern at the type of its position, a tuple component, list element, dictionary value or field,

[2] https://typing.python.org/en/latest/spec/narrowing.html

$\Sigma \vdash C[\vec{\sigma}] \asymp \tau$

instance-above
$$\frac{\Sigma \vdash \tau <: C[\vec{\sigma}] \qquad \vec{\sigma}' \equiv \vec{\sigma} \text{ whenever } \Sigma \vdash \tau <: C[\vec{\sigma}']}{\Sigma \vdash C[\vec{\sigma}] \asymp \tau}$$

instance-object
$$\frac{\text{type-params}_\Sigma(C) = \varepsilon}{\Sigma \vdash C[] \asymp \texttt{object}}$$

instance-below
$$\frac{\text{type-params}_\Sigma(C) = \vec{\alpha} \qquad \text{ancestor-args}_\Sigma(C[\vec{\alpha}], D) = \vec{\tau} \qquad \text{match-args}(\vec{\tau}, \vec{\sigma}') = \gamma \qquad \gamma(\alpha_i) = \sigma_i \quad (\forall i) \qquad \Sigma \vdash C[\vec{\sigma}] <: D[\vec{\sigma}']}{\Sigma \vdash C[\vec{\sigma}] \asymp D[\vec{\sigma}']}$$

(a) $C[\vec{\sigma}]$ is the instance of C at τ (`instance_at`)

$\Sigma \vdash C \parallel \tau$

incomparable-unrelated
$$\frac{C \notin \text{ancestors}_\Sigma(D) \qquad D \notin \text{ancestors}_\Sigma(C)}{\Sigma \vdash C \parallel D[\vec{\sigma}]}$$

incomparable-below
$$\frac{\text{type-params}_\Sigma(C) = \vec{\alpha} \qquad \text{ancestor-args}_\Sigma(C[\vec{\alpha}], D) = \vec{\tau} \qquad \vec{\tau} \# \vec{\sigma}}{\Sigma \vdash C \parallel D[\vec{\sigma}]}$$

incomparable-list
$$\frac{}{\Sigma \vdash C \parallel \texttt{list}[\tau]}$$

incomparable-dict
$$\frac{}{\Sigma \vdash C \parallel \texttt{dict}[\texttt{str}, \tau]}$$

incomparable-tuple
$$\frac{}{\Sigma \vdash C \parallel \texttt{tuple}[\vec{\tau}]}$$

incomparable-callable
$$\frac{}{\Sigma \vdash C \parallel \texttt{Callable}[[\vec{\sigma}], \tau]}$$

incomparable-literal
$$\frac{}{\Sigma \vdash C \parallel \texttt{Literal}[\ell]}$$

incomparable-prim
$$\frac{\nu \notin \{\texttt{object}, \texttt{Never}\}}{\Sigma \vdash C \parallel \nu}$$

(b) Class C is incomparable with τ (`instance_at`)

$\tau \# \sigma$

apart-list
$$\frac{\tau \# \sigma}{\texttt{list}[\tau] \# \texttt{list}[\sigma]}$$

apart-dict
$$\frac{\tau \# \sigma}{\texttt{dict}[\texttt{str}, \tau] \# \texttt{dict}[\texttt{str}, \sigma]}$$

apart-tuple
$$\frac{\vec{\tau} \# \vec{\sigma}}{\texttt{tuple}[\vec{\tau}] \# \texttt{tuple}[\vec{\sigma}]}$$

apart-callable
$$\frac{(\tau \cdot \vec{\tau}) \# (\sigma \cdot \vec{\sigma})}{\texttt{Callable}[[\vec{\tau}], \tau] \# \texttt{Callable}[[\vec{\sigma}], \sigma]}$$

apart-class
$$\frac{\vec{\tau} \# \vec{\sigma}}{C[\vec{\tau}] \# C[\vec{\sigma}]}$$

apart-literal
$$\frac{\ell \neq \ell'}{\texttt{Literal}[\ell] \# \texttt{Literal}[\ell']}$$

apart-differ
$$\frac{\text{neither } \tau \text{ nor } \sigma \text{ a type variable or union} \qquad \tau, \sigma \text{ with different outermost type constructors}}{\tau \# \sigma}$$

$\vec{\tau} \# \vec{\sigma}$

apart-head
$$\frac{\tau \# \sigma}{(\tau \cdot \vec{\tau}) \# (\sigma \cdot \vec{\sigma})}$$

apart-tail
$$\frac{\vec{\tau} \# \vec{\sigma}}{(\tau \cdot \vec{\tau}) \# (\sigma \cdot \vec{\sigma})}$$

apart-length
$$\frac{|\vec{\tau}| \neq |\vec{\sigma}|}{\vec{\tau} \# \vec{\sigma}}$$

(c) Types τ and σ, or sequences $\vec{\tau}$ and $\vec{\sigma}$, are apart

Figure 3.4–1: Instance of a class, incomparability and apartness

$\Sigma; \Gamma \vdash p \lhd \tau \dashv \Delta$

check-var
$$\frac{\Gamma(x) = \mathsf{Unbound}}{\Sigma; \Gamma \vdash x \lhd \tau \dashv \{x : \tau\}}$$

check-wild
$$\frac{}{\Sigma; \Gamma \vdash _ \lhd \tau \dashv \emptyset}$$

check-literal
$$\frac{\tau \text{ has no float values if } \ell \text{ an int literal}}{\Sigma; \Gamma \vdash \ell \lhd \tau \dashv \emptyset}$$

check-as
$$\frac{\Gamma(x) = \mathsf{Unbound} \qquad \Sigma; \Gamma \vdash p \lhd \tau \dashv \Delta}{\Sigma; \Gamma \vdash p \texttt{ as } x \lhd \tau \dashv \Delta \uplus \{x : \text{matched-as}_\Sigma(\Gamma, \tau, p)\}}$$

check-union
$$\frac{\Sigma; \Gamma \vdash p \lhd \sigma \dashv \Delta \qquad \Sigma; \Gamma \vdash p \lhd \tau \dashv \Delta'}{\Sigma; \Gamma \vdash p \lhd \sigma \mid \tau \dashv \{x : \Delta(x) \vee_\Sigma \Delta'(x) \mid x \in \text{binds}(p)\}}$$

check-tuple
$$\frac{\Sigma; \Gamma \vdash p_i \lhd \tau_i \dashv \Delta_i \quad (\forall i)}{\Sigma; \Gamma \vdash (p_1, \ldots, p_n) \lhd \texttt{tuple}[\tau_1, \ldots, \tau_n] \dashv \Delta_1 \uplus \cdots \uplus \Delta_n}$$

check-tuple-none
$$\frac{\tau \text{ has no list or range values} \qquad \text{no disjunct of } \tau \text{ a tuple type of length } |\vec{p}| \qquad \Gamma(x) = \mathsf{Unbound} \quad (\forall x \in \text{binds}(\vec{p}))}{\Sigma; \Gamma \vdash (\vec{p}) \lhd \tau \dashv \{x : \texttt{Never} \mid x \in \text{binds}(\vec{p})\}}$$

check-list
$$\frac{\Sigma; \Gamma \vdash p_i \lhd \tau \dashv \Delta_i \quad (\forall i)}{\Sigma; \Gamma \vdash [\vec{p}] \lhd \texttt{list}[\tau] \dashv \Delta_1 \uplus \cdots \uplus \Delta_n}$$

check-list-none
$$\frac{\tau \text{ has no tuple or range values} \qquad \text{no disjunct of } \tau \text{ a list type} \qquad \Gamma(x) = \mathsf{Unbound} \quad (\forall x \in \text{binds}(\vec{p}))}{\Sigma; \Gamma \vdash [\vec{p}] \lhd \tau \dashv \{x : \texttt{Never} \mid x \in \text{binds}(\vec{p})\}}$$

check-dict
$$\frac{\Sigma; \Gamma \vdash p_i \lhd \tau \dashv \Delta_i \quad (\forall i)}{\Sigma; \Gamma \vdash \{\vec{w} : \vec{p}\} \lhd \texttt{dict}[\texttt{str}, \tau] \dashv \Delta_1 \uplus \cdots \uplus \Delta_n}$$

check-dict-none
$$\frac{\tau \text{ has no dictionary values} \qquad \Gamma(x) = \mathsf{Unbound} \quad (\forall x \in \text{binds}(\vec{p}))}{\Sigma; \Gamma \vdash \{\vec{w} : \vec{p}\} \lhd \tau \dashv \{x : \texttt{Never} \mid x \in \text{binds}(\vec{p})\}}$$

check-constr
$$\frac{\Gamma \vdash q : \mathsf{Type}(C) \qquad \Sigma \vdash C[\vec{\sigma}] \asymp \tau \qquad \text{fields}_\Sigma(C)[\vec{\sigma}] = \vec{x} : \vec{\sigma}' \qquad \text{field-map}_\Sigma(C, a) = \{\vec{x} : \vec{p}\} \qquad \Sigma; \Gamma \vdash p_i \lhd \sigma'_i \dashv \Delta_i \quad (\forall i)}{\Sigma; \Gamma \vdash q(a) \lhd \tau \dashv \Delta_1 \uplus \cdots \uplus \Delta_n}$$

check-constr-none
$$\frac{\Gamma \vdash q : \mathsf{Type}(C) \qquad \Sigma \vdash C \parallel \tau \qquad \Gamma(x) = \mathsf{Unbound} \quad (\forall x \in \text{binds}(q(a)))}{\Sigma; \Gamma \vdash q(a) \lhd \tau \dashv \{x : \texttt{Never} \mid x \in \text{binds}(q(a))\}}$$

Figure 3.4–2: Pattern checking (`check_pat`)

and a pattern at a union at each of its disjuncts. A variable pattern binds at the type at which it is checked, and a pattern named by `as` is bound at matched-as$_\Sigma(\Gamma, \tau, p)$; the bindings of sub-patterns compose by disjoint union, so a pattern that binds a name twice is ill-formed, and at a union the bindings of the disjuncts are joined. A pattern also checks against a type which has no disjunct of the form matched by the pattern: a tuple pattern when no disjunct is a tuple type of the pattern's length, a list pattern when no disjunct is a list type, a dictionary pattern when the type has no dictionary values, or a constructor pattern when its class is incomparable with the type (check-tuple-none, check-list-none, check-dict-none, check-constr-none). Such a pattern matches no shape of the type and binds its variables at `Never`, the unit of the join; as with a variable pattern (check-var), the variables must be unbound in Γ. At one disjunct of a union it therefore leaves the values of that disjunct to later cases, and a case with a pattern matching no shape at all is unreachable.

The conditions on tuple, list and int literal patterns ensure that pattern matching always has a result. Under the evaluation rules, matching a tuple pattern against a list or a list pattern against a tuple, or either against

a range, or an int literal pattern against a float, aborts with excluded (§A.4), although Python permits all of these. Tuple and list patterns therefore fail at `list[int] | tuple[int, int]`, at `Sized | tuple[int, int]`, at `list[int] | builtins.range[]`, and at a type variable. A dictionary pattern at `object` or `Sized`, which have dictionary values, is ill-formed, since the pattern may match a dictionary and yet its variables would be bound at `Never`. The int literal pattern `1` narrows to `Literal[1]`, but Python also matches the pattern against the float `1.0`, which is not a value of `Literal[1]`. An int literal pattern at a type with float values is therefore ill-formed (check-literal).

A constructor pattern cannot name `builtins.range` (pat-constr). Python rejects positional sub-patterns for that class, and PurePy excludes the keyword form as well, for uniformity. The condition on constructor patterns supplies the type arguments omitted by the pattern. The judgement $\Sigma \vdash C[\vec{\sigma}] \asymp \tau$ states that $C[\vec{\sigma}]$ is the *instance* of C at the type τ at which the pattern is checked. A pattern at a union is checked at each disjunct (check-union), so an instance is never taken at a union. Below C the base classes of τ determine the instance, which is unique because type arguments are invariant (instance-above): at `IntLeaf`, a subclass of `Leaf[int]`, the instance of `Leaf` is `Leaf[int]`. Above C the base arguments of C determine the instance (instance-below): at `Tree[int]` the instance of `Leaf` is `Leaf[int]`, because `Leaf[T]` extends `Tree[T]`, but the instance of a class `Node[T, U]` extending `Tree[T]` is undefined, since `U` does not occur in `Tree[T]`. At `object` the instance of a class with type parameters is undefined, since every instantiation lies below `object` (instance-object).

If the class C of a constructor pattern is *incomparable* with τ, no value of type τ is an object of C or of a subclass of C, so the pattern binds its variables at `Never` (check-constr-none). A class C is incomparable with an instantiated class $D[\vec{\sigma}]$ if neither class is an ancestor of the other (incomparable-unrelated), or if the type arguments of the instantiation of D above C applied to its type parameters are *apart* from $\vec{\sigma}$, differing at some position outside type variables and unions, so that no instantiation of the type parameters of C makes the two equivalent (incomparable-below). Constructor patterns are therefore ill-formed at type variables, which may stand for any type. Constructor patterns naming classes with type parameters are also ill-formed at `object` and `Never`. For example, `Node[int, str]` and `Node[int, bool]` both lie below `Tree[int]`, so a pattern naming `Node` is ill-formed at `Tree[int]`.

3.4.2 Shapes and shape typing

A shape k (Figure 3.4–3) denotes a set of values of its type shape-type(k) (§A.2). The form $\tau \setminus H$ denotes the values of type τ whose outermost head is not among those in H; the other forms are constructor, tuple, list and dictionary shapes, each with shapes in place of sub-patterns. A list shape also records its element type, which the element shapes do not determine. A constructor shape also carries an excluded set: $C[\vec{\tau}](\vec{k}) \setminus H$ denotes the instances of $C[\vec{\tau}]$, or of any subclass not below a class of H, whose fields, those of C, have shapes $\vec{k}$. A rest shape at a class type denotes the same values as the constructor shape with a rest shape at each field, and is expanded into that form only by splitting (Figure 3.4–5): a class may refer to itself through its fields, so expanding in advance would not terminate.

A head itself denotes a set of values: a literal its value, a class its instances, and $\texttt{list}_n$ the lists of length n. Head typing $\Sigma \vdash h : \tau$ holds when the values of h lie within τ. Shape typing $\Sigma \vdash k : \tau$ holds when shape-type(k) is a subtype of τ and k is well-formed: each excluded head is typed at the type of the shape (at `str`, for the keys of a dictionary shape) and each component shape at the type of that component.

The auxiliary function $\text{shapes}_\Sigma(\tau, H)$ (§A.2) gives the shapes of τ once the heads in H are excluded. A union contributes the shapes of its disjuncts and a dictionary type the dictionary form; every other type contributes the single shape $\tau \setminus H$, or no shape at all when H exhausts τ. The rest form therefore never carries a union or dictionary type, and shape-rest states this restriction as a side condition.

3.4.3 Matching

Matching (Figure 3.4–4) has three forms. The form on a single shape k matches a pattern against that shape, giving matched shapes K and residual L. The form on sequences matches the sub-patterns of a tuple, list, dictionary or constructor pattern against the component shapes position by position. The form on an ordered set of shapes L, used by the match rules, matches each shape of L in turn. The pattern need not match every

Heads			
h	$::=$	ℓ	(Literal)
	$\mid$	C	(Class)
	$\mid$	$\texttt{list}_n$	(List of length n)
H	$::=$	$\{h_1, \ldots, h_n\}$	(Excluded heads)

Shapes			
k	$::=$	$\tau \setminus H$	(Rest)
	$\mid$	$C[\vec{\tau}](k_1, \ldots, k_n) \setminus H$	(Constr)
	$\mid$	$(k_1, \ldots, k_n)$	(Tuple)
	$\mid$	$[k_1, \ldots, k_n]_\tau$	(List)
	$\mid$	$\beta \setminus_\tau H$	(Dict)
K, L	$::=$	$\{k_1, \ldots, k_n\}$	(Shape set)

(a) Heads and shapes (`head`, `shape`)

$\Sigma \vdash h : \tau$

head-literal
$$\frac{\Sigma \vdash \texttt{Literal}[\ell] <: \tau}{\Sigma \vdash \ell : \tau}$$

head-literal-float
$$\frac{n \text{ a float literal} \qquad \Sigma \vdash \texttt{float} <: \tau}{\Sigma \vdash n : \tau}$$

head-class
$$\frac{\Sigma \vdash C[\vec{\sigma}] <: \tau}{\Sigma \vdash C : \tau}$$

head-list
$$\frac{}{\Sigma \vdash \texttt{list}_n : \texttt{list}[\tau]}$$

(b) Head h has type τ (`head_typed`)

$\Sigma \vdash k : \tau$

shape-rest
$$\frac{\Sigma \vdash h : \tau \quad (\forall h \in H) \qquad \tau \text{ neither union nor dictionary type}}{\Sigma \vdash \tau \setminus H : \tau}$$

shape-constr
$$\frac{\text{fields}_\Sigma(C)[\vec{\tau}] = \vec{x} : \vec{\sigma} \qquad \Sigma \vdash h : C[\vec{\tau}] \quad (\forall h \in H) \qquad \Sigma \vdash k_i : \sigma_i \quad (\forall i)}{\Sigma \vdash C[\vec{\tau}](\vec{k}) \setminus H : C[\vec{\tau}]}$$

shape-tuple
$$\frac{\Sigma \vdash k_i : \tau_i \quad (\forall i)}{\Sigma \vdash (\vec{k}) : \texttt{tuple}[\vec{\tau}]}$$

shape-list
$$\frac{\Sigma \vdash k_i : \tau \quad (\forall i)}{\Sigma \vdash [\vec{k}]_\tau : \texttt{list}[\tau]}$$

shape-sub
$$\frac{\Sigma \vdash k : \sigma \qquad \Sigma \vdash \sigma <: \tau}{\Sigma \vdash k : \tau}$$

shape-dict
$$\frac{\Sigma \vdash h : \texttt{str} \quad (\forall h \in H) \qquad \Sigma \vdash \beta(w) : \tau \quad (\forall w \in \text{dom}(\beta)) \qquad \text{dom}(\beta) \cap H = \emptyset}{\Sigma \vdash \beta \setminus_\tau H : \texttt{dict}[\texttt{str}, \tau]}$$

(c) Shape k has type τ (`shape_typed`)

Figure 3.4–3: Shapes and shape typing

shape of L, so pat-shapes takes the shapes of L matched by the pattern, of which there must be at least one, and passes the unmatched shapes into the residual untouched.

A pattern matches a shape directly if the shape has the head tested by the pattern. Otherwise the shape is first *split* by the auxiliary judgement $\Sigma; \Gamma \vdash k, p \rightsquigarrow_\text{h} K, L$ (Figure 3.4–5): the shapes K have the head tested by p, with the shapes of the component types drawn from shapes_Σ, and the residual L is the shape k with that head excluded as well. The pat-split rule matches the pattern against K and adds L to the residual. Only the forms with an excluded set split, and no head types at a tuple type, so the excluded set of a tuple shape is empty and split-tuple leaves the empty residual. Literal types do not include floats, so a float literal pattern matches a rest shape directly (pat-literal-float) rather than splitting it; the matched shape is the whole shape, and the residual is the shape with the literal excluded. The matched shape therefore includes values which are not equal to the literal. Those values are also in the residual, so the residual of an enclosing pattern is the same as if the matched shape contained only the literal. A constructor pattern whose class lies below a class of the excluded set matches nothing, because an earlier case for the superclass has already matched every instance. A constructor pattern matches the instance of its class at the type of the shape (Figure 3.4–1): a pattern `Leaf(v)` at `Tree[int]` matches `Leaf[int]` and binds `v` at `int`.

$\Sigma; \Gamma \vdash k, p \rightsquigarrow K, L$

pat-var

$$\frac{}{\Sigma; \Gamma \vdash k, x \rightsquigarrow \{k\}, \emptyset}$$

pat-wild

$$\frac{}{\Sigma; \Gamma \vdash k, _ \rightsquigarrow \{k\}, \emptyset}$$

pat-as

$$\frac{\Sigma; \Gamma \vdash k, p \rightsquigarrow K, L}{\Sigma; \Gamma \vdash k, p \texttt{ as } x \rightsquigarrow K, L}$$

pat-split

$$\frac{\Sigma; \Gamma \vdash k, p \rightsquigarrow_{\mathsf{h}} K, L \qquad \Sigma; \Gamma \vdash K, p \rightsquigarrow K', L'}{\Sigma; \Gamma \vdash k, p \rightsquigarrow K', L \uplus L'}$$

pat-literal

$$\frac{}{\Sigma; \Gamma \vdash \texttt{Literal}[\ell] \setminus \emptyset, \ell \rightsquigarrow \{\texttt{Literal}[\ell] \setminus \emptyset\}, \emptyset}$$

pat-literal-float

$$\frac{n \text{ a float literal} \qquad n \notin H \qquad \Sigma \vdash \texttt{float} <: \tau}{\Sigma; \Gamma \vdash \tau \setminus H, n \rightsquigarrow \{\tau \setminus H\}, \{\tau \setminus (H \cup \{n\})\}}$$

pat-tuple

$$\frac{\Sigma; \Gamma \vdash \vec{k}, \vec{p} \rightsquigarrow K, L}{\Sigma; \Gamma \vdash (\vec{k}), (\vec{p}) \rightsquigarrow \{(\vec{k}') \mid \vec{k}' \in K\}, \{(\vec{l}) \mid \vec{l} \in L\}}$$

pat-list

$$\frac{\Sigma; \Gamma \vdash \vec{k}, \vec{p} \rightsquigarrow K, L}{\Sigma; \Gamma \vdash [\vec{k}]_\tau, [\vec{p}] \rightsquigarrow \{[\vec{k}']_\tau \mid \vec{k}' \in K\}, \{[\vec{l}]_\tau \mid \vec{l} \in L\}}$$

pat-dict

$$\frac{\vec{w} \text{ distinct} \qquad k_i = \beta(w_i) \quad (\forall i) \qquad \Sigma; \Gamma \vdash \vec{k}, \vec{p} \rightsquigarrow K, L}{\Sigma; \Gamma \vdash \beta \setminus_\tau H, \{\vec{w} : \vec{p}\} \rightsquigarrow \{\beta \otimes \{\vec{w} : \vec{k}'\} \setminus_\tau H \mid \vec{k}' \in K\}, \{\beta \otimes \{\vec{w} : \vec{l}\} \setminus_\tau H \mid \vec{l} \in L\}}$$

pat-constr

$$\frac{\begin{array}{c}\Gamma \vdash q : \mathsf{Type}(D) \qquad D \neq \texttt{builtins.range} \qquad D \in \mathrm{ancestors}_\Sigma(C) \\ \mathrm{fields}_\Sigma(D) = \forall \vec{\alpha}. \vec{x} : \vec{\sigma} \qquad \text{field-map}_\Sigma(D, a) = \{\vec{x} : \vec{p}\} \qquad p_i' = _ \quad (\forall i) \qquad \Sigma; \Gamma \vdash \vec{k}, \vec{p} +\!\!+ \vec{p}' \rightsquigarrow K, L\end{array}}{\Sigma; \Gamma \vdash C[\vec{\tau}](\vec{k}) \setminus H, q(a) \rightsquigarrow \{C[\vec{\tau}](\vec{k}') \setminus H \mid \vec{k}' \in K\}, \{C[\vec{\tau}](\vec{l}) \setminus H \mid \vec{l} \in L\}}$$

(a) Pattern p matches shape k, giving matched shapes K and residual L (`match_shape`)

$\Sigma; \Gamma \vdash \vec{k}, \vec{p} \rightsquigarrow K, L$

pat-nil

$$\frac{}{\Sigma; \Gamma \vdash \varepsilon, \varepsilon \rightsquigarrow \{\varepsilon\}, \emptyset}$$

pat-cons

$$\frac{\Sigma; \Gamma \vdash k, p \rightsquigarrow K, L \qquad \Sigma; \Gamma \vdash \vec{k}, \vec{p} \rightsquigarrow K', L'}{\Sigma; \Gamma \vdash k \cdot \vec{k}, p \cdot \vec{p} \rightsquigarrow K \times K', (L \times \{\vec{k}\}) \uplus (K \times L')}$$

(b) Patterns $\vec{p}$ match shapes $\vec{k}$, giving matched shapes K and residual L

$\Sigma; \Gamma \vdash L, p \rightsquigarrow K, L'$

pat-shapes

$$\frac{L' = \{k \in L \mid \Sigma; \Gamma \vdash k, p \rightsquigarrow K_k, L_k\} \qquad L' \neq \emptyset}{\Sigma; \Gamma \vdash L, p \rightsquigarrow \biguplus_{k \in L'} K_k, \ \biguplus_{k \in L'} L_k \uplus (L \setminus L')}$$

(c) Pattern p matches ordered set of shapes L, giving matched shapes K and residual L' (`match_shapes`)

Figure 3.4–4: Matching patterns against shapes

$\Sigma;\Gamma \vdash k, p \rightsquigarrow_{\mathsf{h}} K, L$

split-literal

$$\frac{\ell \notin H \qquad \Sigma \vdash \texttt{Literal}[\ell] <: \tau \qquad \tau \text{ not a literal type}}{\Sigma;\Gamma \vdash \tau \setminus H, \ell \rightsquigarrow_{\mathsf{h}} \{\texttt{Literal}[\ell] \setminus \emptyset\}, \mathrm{shapes}_{\Sigma}(\tau, H \cup \{\ell\})}$$

split-tuple

$$\frac{|\vec{\tau}| = |\vec{p}|}{\Sigma;\Gamma \vdash \texttt{tuple}[\vec{\tau}] \setminus \emptyset, (\vec{p}) \rightsquigarrow_{\mathsf{h}} \{(\vec{k}) \mid \vec{k} \in \mathrm{shapes}_{\Sigma}(\vec{\tau})\}, \emptyset}$$

split-list

$$\frac{n = |\vec{p}| \qquad \texttt{list}_n \notin H}{\Sigma;\Gamma \vdash \texttt{list}[\tau] \setminus H, [\vec{p}] \rightsquigarrow_{\mathsf{h}} \{[\vec{k}]_\tau \mid \vec{k} \in \mathrm{shapes}_{\Sigma}(\tau, \emptyset)^n\}, \mathrm{shapes}_{\Sigma}(\texttt{list}[\tau], H \cup \{\texttt{list}_n\})}$$

split-dict

$$\frac{w_i \notin \mathrm{dom}(\beta) \cup H \qquad w_j \in \mathrm{dom}(\beta) \quad (\forall j < i)}{\Sigma;\Gamma \vdash \beta \setminus_\tau H, \{\vec{w} : \vec{p}\} \rightsquigarrow_{\mathsf{h}} \{\beta \otimes \{w_i : k\} \setminus_\tau H \mid k \in \mathrm{shapes}_{\Sigma}(\tau, \emptyset)\}, \{\beta \setminus_\tau (H \cup \{w_i\})\}}$$

split-class

$$\frac{\Gamma \vdash q : \mathsf{Type}(C) \qquad \mathrm{ancestors}_{\Sigma}(C) \cap H = \emptyset \qquad \Sigma \vdash C[\vec{\tau}'] \asymp \tau \qquad \tau \wedge_{\Sigma} C[\vec{\tau}'] = D[\vec{\sigma}] \qquad \mathrm{fields}_{\Sigma}(D)[\vec{\sigma}] = \vec{x} : \vec{\sigma}' \qquad H' = \{h \in H \mid \Sigma \vdash h : D[\vec{\sigma}]\}}{\Sigma;\Gamma \vdash \tau \setminus H, q(a) \rightsquigarrow_{\mathsf{h}} \{D[\vec{\sigma}](\vec{k}) \setminus H' \mid \vec{k} \in \mathrm{shapes}_{\Sigma}(\vec{\sigma}')\}, \mathrm{shapes}_{\Sigma}(\tau, H \cup \{C\})}$$

split-subclass

$$\frac{\Gamma \vdash q : \mathsf{Type}(D) \qquad \Sigma \vdash D[\vec{\sigma}] \asymp C[\vec{\tau}] \qquad \Sigma \vdash D[\vec{\sigma}] <: C[\vec{\tau}] \qquad D \neq C \qquad \mathrm{ancestors}_{\Sigma}(D) \cap H = \emptyset \qquad \mathrm{fields}_{\Sigma}(D)[\vec{\sigma}] = \mathrm{fields}_{\Sigma}(C)[\vec{\tau}] +\!\!+ \vec{y} : \vec{\sigma}' \qquad H' = \{h \in H \mid \Sigma \vdash h : D[\vec{\sigma}]\}}{\Sigma;\Gamma \vdash C[\vec{\tau}](\vec{k}) \setminus H, q(a) \rightsquigarrow_{\mathsf{h}} \{D[\vec{\sigma}](\vec{k} +\!\!+ \vec{k}') \setminus H' \mid \vec{k}' \in \mathrm{shapes}_{\Sigma}(\vec{\sigma}')\}, \{C[\vec{\tau}](\vec{k}) \setminus (H \cup \{D\})\}}$$

Figure 3.4–5: Shape k splits at the head tested by p, giving shapes K and residual L (`split`)

A dictionary shape has a separate form, $\beta \setminus_\tau H$, standing for the dictionaries that hold each key w in the domain of β with a value of shape $\beta(w)$, but that hold no key of H. A dictionary pattern takes its keys in order, with no key occurring twice. A key in neither β nor H splits the shape at that key (split-dict), adding to the residual the dictionaries that lack the key; once every key of the pattern is in β the keys match as a sequence (pat-dict), each key w against $\beta(w)$. The empty dictionary pattern matches every dictionary and leaves the empty residual.

The cases needed to exhaust a match therefore depend on the type of the scrutinee: a union is exhausted by covering each disjunct, a class type by a pattern for the class itself, `bool` by `True` and `False`, and a dictionary type by the empty dictionary pattern, while `int`, `float`, `str` and lists have unboundedly many heads and need a variable or wildcard. A literal-typed scrutinee is exhausted by covering its literals, so `x: Literal[42] = 42` admits a match with a case for `42` alone, while `x: int = 42` keeps the type open. Over `tuple[bool, bool]` the cases `(True, _)`, `(_, False)` and `(False, True)` leave the empty residual, although each covers only part of the type.

3.5 Modules and programs

Module checking types a module's body and computes its signature Γ, the judgement $\Sigma \vdash_{\mathsf{mod}} q : \Gamma \dashv \Sigma'$ (Figure 3.5–3a); it mirrors loading at run time (Chapter 4). The module rule types the body as a top-level statement, and the predefined rule gives a predefined module the signature of Definition 2.7. Both import forms check every module containing the target of the import (Figure 3.5–2). A plain import checks the root and then each module along the chain of names, recording each as a member of its parent (submodules-cons), so that the binding for the root gives attribute access along the whole chain. Two imports with the same root, such as `a.b` and `a.c`, both bind the root, and their bindings combine by *context extension* $\Gamma \boxplus \Delta$ rather than shadow one another. Extension is defined pointwise, like override, and combines module entries for the same module recursively, preferring checked modules to stubs:

Definition 3.5 (Extension of context entries)**.**

$$\begin{aligned}\mathsf{Mod}(q,\Gamma) \boxplus \mathsf{Mod}(q,\Gamma') &= \mathsf{Mod}(q,\Gamma \boxplus \Gamma')\\ \mathsf{Mod}(q) \boxplus \mathsf{Mod}(q,\Gamma) = \mathsf{Mod}(q,\Gamma) \boxplus \mathsf{Mod}(q) &= \mathsf{Mod}(q,\Gamma)\\ \bot \boxplus \theta = \theta \boxplus \bot &= \theta\\ \theta \boxplus \theta' &= \theta' \quad \text{(otherwise)}\end{aligned}$$

A from-import also checks any imported name that denotes a submodule, and the names it imports must be distinct. A module which strictly contains the target is exempt from checking if it contains the importing module (§A.3), because at runtime it may still be loading when the import executes. The exemption applies only to from-imports, because for a plain import a module which strictly contains the target can still be loading only if it is the importing module, and a plain import of a module contained in the importing module is excluded outright.

Module checking is defined inductively, so a program with an import cycle is ill-formed. Program well-formedness (Figure 3.5–3b) requires the main module to be well typed and hence, recursively, every module that the program can load; a module that is never imported is not checked. The rule module types a module body under the context of $\mathcal{P}$(`builtins`), overridden by the imported context and then, as in def-body, by an unbound entry for each name declared in the body or bound by a pattern in it. The members of `builtins` are thus in scope unqualified, and a name bound by the body shadows a builtin throughout the module, so a use of the name before its declaration is rejected. Under mypy such a use instead resolves to the builtin, following Python's run-time lookup. As with any scope, the names declared in the module, including its imported names, must be distinct from one another and from any pattern variables in its body. Since every module declares `__name__` (§2.4), a module is ill-formed if its body declares that name again, imports it, or binds it in a pattern. The declaration of `__name__` is annotated with `str`, so a module is also ill-formed if its body binds `str`.

$\Sigma \vdash \Gamma, x \text{ imports } \theta \dashv \Sigma'$

imp-member

$$\frac{\Gamma \vdash x : \theta}{\Sigma \vdash \Gamma, x \text{ imports } \theta \dashv \Sigma}$$

imp-submodule

$$\frac{x\colon \mathsf{Mod}(q) \in \Gamma \qquad \Sigma \vdash_{\mathsf{mod}} q : \Delta \dashv \Sigma'}{\Sigma \vdash \Gamma, x \text{ imports } \mathsf{Mod}(q,\Delta) \dashv \Sigma'}$$

Figure 3.5–1: Name x imported from module signature Γ has context entry θ (`imports_entry`)

The signature Γ produced by module checking (Figure 3.5–3a) comprises the classes, type aliases and variables defined by a module, together with its intrinsic `__name__` and a stub for each submodule, but not the names that it imports: following PEP 484[3], an imported name is a private dependency, not re-exported as part of the interface. Submodule stubs and the bindings of the module body are disjoint, because the module rules exclude a module that binds a name that is also the name of a submodule.

[3] https://peps.python.org/pep-0484/

$\Sigma; q \vdash \vec{\imath} : \Gamma \dashv \Sigma'$

imports-nil

$$\frac{}{\Sigma; q \vdash \varepsilon : \emptyset \dashv \Sigma}$$

import

$$\frac{q' = x \cdot \vec{x} \qquad q \not\prec q' \qquad \Sigma \vdash_{\mathsf{mod}} x : \Delta \dashv \Sigma_1 \qquad \Sigma_1 \vdash_{\mathsf{sub}} x, \Delta, \vec{x} : \Delta' \dashv \Sigma_2 \qquad \Sigma_2; q \vdash \vec{\imath} : \Gamma \dashv \Sigma'}{\Sigma; q \vdash \texttt{import}\ q' \cdot \vec{\imath} : \{x : \mathsf{Mod}(x, \Delta')\} \boxplus \Gamma \dashv \Sigma'}$$

from-import

$$\frac{\begin{array}{c} x_1, \ldots, x_{n+1}\ \text{distinct} \qquad \Sigma \vdash_{\mathsf{mod}} q' : \Delta \dashv \Sigma_0 \qquad q_1^\dagger \cdot \cdots \cdot q_m^\dagger = \text{unique-to}(q', q) \\ \Sigma_{j-1} \vdash_{\mathsf{mod}} q_j^\dagger : \Delta_j \dashv \Sigma_j \quad (\forall j) \qquad \Sigma_{m+i-1} \vdash \Delta, x_i\ \mathsf{imports}\ \theta_i \dashv \Sigma_{m+i} \quad (\forall i) \qquad \Sigma_{m+n+1}; q \vdash \vec{\imath} : \Gamma \dashv \Sigma' \end{array}}{\Sigma; q \vdash \texttt{from}\ q'\ \texttt{import}\ x_1, \ldots, x_{n+1} \cdot \vec{\imath} : \{x_1 : \theta_1, \ldots, x_{n+1} : \theta_{n+1}\} \boxplus \Gamma \dashv \Sigma'}$$

(a) Import prefix $\vec{\imath}$ of module q produces bindings Γ (`check_imports`)

$\Sigma \vdash_{\mathsf{sub}} q, \Gamma, \vec{x} : \Gamma' \dashv \Sigma'$

submodules-nil

$$\frac{}{\Sigma \vdash_{\mathsf{sub}} q, \Gamma, \varepsilon : \Gamma \dashv \Sigma}$$

submodules-cons

$$\frac{\Sigma \vdash_{\mathsf{mod}} q.x : \Delta \dashv \Sigma^\dagger \qquad \Sigma^\dagger \vdash_{\mathsf{sub}} q.x, \Delta, \vec{x} : \Delta' \dashv \Sigma'}{\Sigma \vdash_{\mathsf{sub}} q, \Gamma, x \cdot \vec{x} : \Gamma \boxplus \{x : \mathsf{Mod}(q.x, \Delta')\} \dashv \Sigma'}$$

(b) Submodule chain $\vec{x}$ extends signature Γ of module q to Γ' (`submodules`)

Figure 3.5–2: Import typing

$\Sigma \vdash_{\mathsf{mod}} q : \Gamma \dashv \Sigma'$

predefined

$$\frac{\mathcal{P}(q) = \Gamma}{\Sigma \vdash_{\mathsf{mod}} q : \Gamma \dashv \Sigma}$$

module

$$\frac{\begin{array}{c} \mathcal{M}(q) = \vec{\imath}\ t \qquad \mathcal{P}(\texttt{builtins}) = \Gamma \qquad \Sigma; q \vdash \vec{\imath} : \Gamma' \dashv \Sigma^\dagger \qquad \Sigma^\dagger; q; \Gamma \otimes \Gamma' \otimes \text{scope}(\text{declares}(\vec{\imath}), t) \vdash t : \mathsf{Assigns}(\Delta) \dashv \Sigma' \\ (\text{dom}(\Gamma') \cup \text{dom}(\text{scope}(\text{declares}(\vec{\imath}), t))) \cap \text{dom}(\text{submods}(q)) = \emptyset \end{array}}{\Sigma \vdash_{\mathsf{mod}} q : \text{submods}(q) \otimes \Delta \dashv \Sigma'}$$

(a) Module q checks with signature Γ (`check_module`)

$\vdash \mathcal{M}$

program

$$\frac{\Sigma_{\mathcal{P}} \vdash_{\mathsf{mod}} \texttt{__main__} : \Gamma \dashv \Sigma}{\vdash \mathcal{M}}$$

(b) Program $\mathcal{M}$ is well-formed (`check_program`)

Figure 3.5–3: Module typing

4 Operational semantics

4.1 Values and environments

Values, environments, outcomes and results are given in Figure 4.1–1. A class value $\mathsf{Type}(C)$ is the context entry of the class, and an object $\mathsf{Obj}(C, \vec{x} : \vec{v})$ records its class by name and fields in the order given by $\mathrm{fields}_\Sigma(C)$. The value $\mathsf{Predef}(q.x)$ represents a predefined member by its name alone, such as the class `int` or decorator `dataclass`. The value $\mathsf{Unbound}$ marks a variable of the current scope before its declaration is reached, which Python reports as an unbound local. An environment binds a variable to $\mathsf{Unbound}$ wherever the corresponding context binds the variable to the entry $\mathsf{Unbound}$, so that a shadowed binding of an enclosing scope is not visible. Throughout this section, Σ is the class table produced by module checking of the program (§3.5) and, like $\mathcal{M}$ and $\mathcal{P}$, is ambient. For a statement s, outcome $(\vec{v}, \omega)$ represents s printing the values $\vec{v}$, in order, and completing as ω. Completion $\mathsf{assigns}(\rho)$ represents s completing normally, with accumulated environment ρ; completion $\mathsf{returns}(v)$ represents s returning the value v; and completion $\mathsf{aborts}(\kappa)$ represents the whole run aborting, with termination kind κ.

Definition 4.1 (Composition of outcomes)**.** Composing statements sequentially composes their outcomes according to the following monoid, analogous to sequential composition of static outcomes:

$$
\begin{aligned}
(\vec{v}, \mathsf{returns}(v)) \otimes o &= (\vec{v}, \mathsf{returns}(v)) \\
(\vec{v}, \mathsf{aborts}(\kappa)) \otimes o &= (\vec{v}, \mathsf{aborts}(\kappa)) \\
(\vec{v}, \mathsf{assigns}(\rho)) \otimes (\vec{v}', \mathsf{assigns}(\rho')) &= (\vec{v} +\!\!+ \vec{v}', \mathsf{assigns}(\rho \otimes \rho')) \\
(\vec{v}, \mathsf{assigns}(\rho)) \otimes (\vec{v}', \omega) &= (\vec{v} +\!\!+ \vec{v}', \omega) \quad \text{(otherwise)}
\end{aligned}
$$

with outcomes of the form $(\vec{v}, \mathsf{returns}(v))$ or $(\vec{v}, \mathsf{aborts}(\kappa))$ as left zeros, and $(\varepsilon, \mathsf{assigns}(\emptyset))$ as left and right unit.

Termination kinds κ record how a run ends; all but undef and excluded are named after the exceptions raised by the same program under Python in the same circumstances, except that $\mathsf{SystemExit}(0)$ also records a run which completes normally. A variable read before its assignment aborts with $\mathsf{NameError}$ where Python raises `UnboundLocalError`, a subclass of `NameError`; distinguishing the two would require the environment to record whether unbound variables belong to functions or modules. A run aborts with undef when it reaches a dynamically excluded operation, one which the type system cannot rule out and PurePy leaves undefined, such as a comparison of two functions, or an operation whose outcome is not specified by the Python language reference and which differs between implementations, of which an overflowing $**$ is the only example at present (§B.1.3). A run aborts with excluded when it reaches an operation ruled out by the type system, such as a condition which evaluates to a non-Boolean. Attribute references, dictionary keys and constructor patterns headed by a built-in class, when ruled out by the type system, also abort with excluded, even though Python sometimes rejects them; to be more precise here would require the semantics to model the attributes Python assigns to a value, the keys which Python considers hashable, and whether a value is an instance of an opaque class such as `int` (§4.3, §4.4). A primitive $\mathsf{Prim}(f)$ is a predefined function whose meaning f is a function from arguments to results (Figure 2.4–2). Calling $\mathsf{Prim}(f)$ with arguments $\vec{v}$ has result $f(\vec{v})$.

Rule conclusions use a monadic let notation over results. If r is $\mathsf{val}(v)$, then $\mathsf{let}\ v \leftarrow r\ \mathsf{in}\ \ldots$ is the body with v bound to that value; if r is $\mathsf{aborts}(\kappa)$, it is $\mathsf{aborts}(\kappa)$ as a result, or $(\varepsilon, \mathsf{aborts}(\kappa))$ as an outcome. The body is a result in the expression rules and an outcome in the statement rules. A final premise needs no propagation rule; earlier premises have one each, recording the subexpressions evaluated before the failing one (Figure 4.3–6). The sequence judgement uses results carrying a sequence of values in place of a value.

Values			
v	::=	ℓ	(Literal)
	|	$[v_1, \ldots, v_n]$	(List)
	|	$(v_1, \ldots, v_n)$	(Tuple)
	|	$\{\delta\}$	(Dict)
	|	$\mathsf{Lam}_\Gamma(\rho, \vec{x}, e)$	(LambdaClosure)
	|	$\mathsf{Def}_\Gamma(\rho, \vec{d}, i)$	(DefClosure)
	|	$\mathsf{Obj}(C, x_1 : v_1 \cdot \ldots \cdot x_n : v_n)$	(Object)
	|	$\mathsf{Mod}(q)$	(ModuleStub)
	|	$\mathsf{Mod}(q, \rho)$	(LoadedModule)
	|	$\mathsf{Type}(C)$	(Class)
	|	$\mathsf{TypeAlias}_\Gamma(q, \vec{\chi}, i)$	(TypeAlias)
	|	Unbound	(Unbound)
	|	$\mathsf{Prim}(f)$	(Primitive)
	|	$\mathsf{Predef}(q.x)$	(Predef)

Environments		
ρ	::=	$\{x_1 : v_1, \ldots, x_n : v_n\}$

Dictionary entries		
δ	::=	$w_1 : v_1 \cdot \ldots \cdot w_n : v_n$

Outcomes			
o	::=	$(\vec{v}, \omega)$	
ω	::=	$\mathsf{assigns}(\rho)$	(Assigns)
	|	$\mathsf{returns}(v)$	(Returns)
	|	$\mathsf{aborts}(\kappa)$	(Aborts)

Results			
r	::=	$\mathsf{val}(v)$	(Val)
	|	$\mathsf{aborts}(\kappa)$	(Aborts)

Match results			
m	::=	$\mathsf{match}(\rho)$	(Match)
	|	no-match	(NoMatch)
	|	κ	(Kind)

Termination kinds		
κ	::=	TypeError | IndexError | KeyError
	|	OverflowError | ValueError | ZeroDivisionError
	|	AssertionError | $\mathsf{AssertionError}(w)$ | AttributeError
	|	NameError | $\mathsf{SystemExit}(n)$ | undef | excluded

Figure 4.1–1: Values, environments, outcomes and results (`val`, `env`, `outcome`, `result`)

4.2 Auxiliary definitions

The operators and the other auxiliary functions the evaluation rules use are defined in §A.4, and module and class resolution in §A.3.

4.2.1 Operators

The implementation of an operator is selected at run time by the run-time types of its operands, using the run-time overloads of Figure 4.2–1. The function type-of(v) (§A.4) gives the run-time type of a value. Float arithmetic follows IEEE 754, except that `//` and `%` on floats are defined by floordiv and mod (§A.4), in agreement with the Python Language Reference, §6.7. The operator `**` on floats gives the exact power rounded by fl,

$\odot$	(τ_1, τ_2)	$\hat{\odot}(v, v')$
`==`	(`object`, `object`)	val(`False`) if $v = v' = \mathsf{nan}$; $\mathrm{eq}_\Sigma(v, v')$ otherwise
`!=`	(`object`, `object`)	val(`True`) if $v = v' = \mathsf{nan}$ or $\mathrm{eq}_\Sigma(v, v') = \mathsf{val}(\texttt{False})$; val(`False`) if $\mathrm{eq}_\Sigma(v, v') = \mathsf{val}(\texttt{True})$; $\mathrm{eq}_\Sigma(v, v')$ otherwise
`in`	(`object`, `object`)	$\mathrm{in}_\Sigma(v, v')$
`not in`	(`object`, `object`)	val(`True`) if $\mathrm{in}_\Sigma(v, v') = \mathsf{val}(\texttt{False})$; val(`False`) if $\mathrm{in}_\Sigma(v, v') = \mathsf{val}(\texttt{True})$; $\mathrm{in}_\Sigma(v, v')$ otherwise
`<`	(`float`, `float`)	val(`True`) if $\mathrm{real}(v) < \mathrm{real}(v')$; val(`False`) otherwise
	(`str`, `str`)	val(`True`) if $v < v'$; val(`False`) otherwise
`<=`	(`float`, `float`)	val(`True`) if $\mathrm{real}(v) \leq \mathrm{real}(v')$; val(`False`) otherwise
	(`str`, `str`)	val(`True`) if $v \leq v'$; val(`False`) otherwise
`>`	(`float`, `float`)	val(`True`) if $\mathrm{real}(v) > \mathrm{real}(v')$; val(`False`) otherwise
	(`str`, `str`)	val(`True`) if $v > v'$; val(`False`) otherwise
`>=`	(`float`, `float`)	val(`True`) if $\mathrm{real}(v) \geq \mathrm{real}(v')$; val(`False`) otherwise
	(`str`, `str`)	val(`True`) if $v \geq v'$; val(`False`) otherwise
`+`	(`int`, `int`)	$\mathsf{val}(v + v')$
	(`float`, `float`)	$\mathsf{val}(v + v')$
	(`str`, `str`)	$\mathsf{val}(v +\!\!+ v')$
	(`list`[`object`], `list`[`object`])	$\mathsf{val}([\vec{v} +\!\!+ \vec{v}'])$ where $v = [\vec{v}]$ and $v' = [\vec{v}']$
	(`tuple`[`object`^{m}], `tuple`[`object`^{n}])	$\mathsf{val}((\vec{v} +\!\!+ \vec{v}'))$ where $v = (v_1, \ldots, v_m)$ and $v' = (v'_1, \ldots, v'_n)$
`-`	(`int`, `int`)	$\mathsf{val}(v - v')$
	(`float`, `float`)	$\mathsf{val}(v - v')$
`*`	(`int`, `int`)	$\mathsf{val}(v \times v')$
	(`float`, `float`)	$\mathsf{val}(v \times v')$
	(`str`, `int`)	$\mathsf{val}(+\!\!+(v^n))$ where $n = \max(v', 0)$
	(`int`, `str`)	$\mathsf{val}(+\!\!+(v'^n))$ where $n = \max(v, 0)$
	(`list`[`object`], `int`)	$\mathsf{val}([+\!\!+(\vec{v}^n)])$ where $v = [\vec{v}]$ and $n = \max(v', 0)$
	(`int`, `list`[`object`])	$\mathsf{val}([+\!\!+(\vec{v}^n)])$ where $v' = [\vec{v}]$ and $n = \max(v, 0)$
`/`	(`int`, `int`)	$\mathrm{float}(v/v')$ if $v' \neq 0$; aborts(ZeroDivisionError) otherwise
	(`float`, `float`)	$\mathsf{val}(v/v')$ if $\mathrm{real}(v') \neq 0$; aborts(ZeroDivisionError) otherwise
`//`	(`int`, `int`)	$\mathsf{val}(\lfloor v/v' \rfloor)$ if $v' \neq 0$; aborts(ZeroDivisionError) otherwise
	(`float`, `float`)	$\mathsf{val}(\mathrm{floordiv}(v, v'))$ if $\mathrm{real}(v') \neq 0$; aborts(ZeroDivisionError) otherwise
`%`	(`int`, `int`)	$\mathsf{val}(v - (v' \times \lfloor v/v' \rfloor))$ if $v' \neq 0$; aborts(ZeroDivisionError) otherwise
	(`float`, `float`)	$\mathsf{val}(\mathrm{mod}(v, v'))$ if $\mathrm{real}(v') \neq 0$; aborts(ZeroDivisionError) otherwise
`**`	(`int`, `int`)	$\mathsf{val}(v^{v'})$ if $v' \geq 0$; aborts(ZeroDivisionError) if $v = 0$ and $v' < 0$; as at (`float`, `int`) otherwise
	(`float`, `int`)	aborts(ZeroDivisionError) if $\mathrm{real}(v) = 0$ and $v' < 0$; aborts(undef) if v finite and $\mathrm{fl}(\mathrm{real}(v)^{v'})$ infinite; $\mathsf{val}(\mathrm{fl}(\mathrm{real}(v)^{v'}))$ otherwise

$\ominus$	τ	$\hat{\ominus}(v)$
`not`	`bool`	val(`False`) if $v =$ `True`; val(`True`) otherwise
`+`	`float`	$\mathsf{val}(v)$
`-`	`float`	$\mathsf{val}(-v)$

Figure 4.2–1: Run-time overloads of operators (`overloads`)

although a conforming implementation may round it differently (§4.5). The comparisons `<`, `>`, `<=` and `>=` compare strings lexicographically by code point. An integer n passed to an arithmetic operator which expects a float is converted using float(n), and the operator aborts if the conversion aborts. The functions fl and float (§A.4) both round a real number to a float; fl is a shorthand for rounding attribute roundTiesToEven of IEEE 754, and float aborts with OverflowError when fl gives an infinity.

Definition 4.2 (Run-time overload; `overload`)**.** A *run-time overload* of a binary operator $\odot$ pairs operand bounds (τ_1, τ_2) with a definition of $\hat{\odot}(v, v')$, and applies at operand types (σ_1, σ_2) with $\Sigma \vdash \sigma_1 <: \tau_1$ and $\Sigma \vdash \sigma_2 <: \tau_2$. Run-time overloads are ordered by componentwise subtyping in Σ on their bounds. Run-time overloads of a unary operator $\ominus$ pair an operand bound τ with a definition of $\hat{\ominus}(v)$ in the same way.

The result $\hat{\odot}(v, v')$ is given by the most specific run-time overload of $\odot$ at (type-of(v), type-of(v')), and is aborts(excluded) if no run-time overload of $\odot$ applies. The result $\hat{\ominus}(v)$ is given in the same way. Given x : `float` = 1, the expression x + 1 has type `float` but evaluates to 2 via the run-time overload at (`int`, `int`).

Lemma 4.1 (Soundness of overload resolution; `binop_meaning_typed`, `unop_meaning_typed`)**.** *Suppose* $\vdash v : \sigma_1$ *and* $\vdash v' : \sigma_2$ *with* resolve-op$_\Sigma(\odot, \sigma_1, \sigma_2) = \tau$*. Then some run-time overload of* $\odot$ *applies at* (type-of(v), type-of(v'))*. Moreover, if* $\hat{\odot}(v, v') =$ val($v^\dagger$) *then* $\vdash v^\dagger : \tau$*, and* $\hat{\odot}(v, v')$ *does not abort with* TypeError*,* AttributeError*,* NameError *or* excluded*. The analogous statement applies for* $\ominus$*.*

4.3 Expressions and statements

The value typing judgement $\vdash v : \theta$ says that value v is well-typed at context entry θ, and the environment typing judgement $\vdash \rho : \Gamma$ says that environment ρ is well-typed at context Γ, meaning every value in ρ is well-typed at the corresponding entry in Γ (Figure 4.3–1). Each rule typing a value at a type has a subtyping premise, so at most one rule applies to a given value and context entry. The val-def rule instantiates the type scheme of a function at closed types, matching the instantiation of names bound at a scheme by the var-scheme rule. The statement evaluation judgement $\rho, s \Rightarrow o$ says that s evaluates to outcome o in ρ (Figures 4.3–2 and 4.3–3). An `if` or `assert` statement aborts with excluded when a condition evaluates to a non-Boolean (eval-if-nonbool, eval-if-else-nonbool, eval-assert-nonbool, eval-assert-msg-nonbool), and a failing `assert` statement also aborts with excluded when its message is not a string (eval-assert-msg-nonstr). Top-level statements are evaluated by a separate judgement, $q, \rho, t \Rightarrow o$ (Figure 4.3–4), mirroring the static judgement (Figure 3.3–5): a sequence composes in the same way as a statement sequence, a mutual type region binds each of its names to type-entry$_\Gamma(q, \vec{\chi}, i)$ (eval-types), since class and alias entries are also values, and a plain statement is evaluated by the statement judgement (eval-top-stmt). Function definitions delimit the scope of sequential-composition effects, so as black boxes functions are always pure. The auxiliary function result (§A.4) reads the outcome of a statement as the result of a call; if the statement completes without returning, the result is val(`None`).

The expression evaluation judgement $\rho, e \Rightarrow r$ (Figures 4.3–5 and 4.3–7) says that e evaluates to result r in ρ, where r is either val(v) or aborts(κ) (Figure 4.1–1). A variable not bound in ρ, or bound to Unbound, aborts with NameError (eval-var-unknown, eval-var-unbound). Function calls and constructor calls share one syntactic form, as do subscripting a container and applying a class to type arguments; the semantics distinguishes them by the value of the callee or subject. A call whose callee evaluates to a class is a constructor call, and a subscript expression whose subject evaluates to a class evaluates to the class itself, since values carry no type arguments (eval-constr, eval-class-type-args). A call of a value other than a closure, primitive or class, a call with the wrong number of arguments, and a constructor call with arguments which do not cover the fields of the class, abort with TypeError (eval-call-nonfun, eval-call-lambda-arity, eval-call-def-arity, eval-constr-arity), and a reference to an attribute absent from an object or a loaded module aborts with AttributeError (eval-attr-object-missing, eval-attr-module-missing), as in Python. The operators `and` and `or` evaluate their second operand only when the first leaves the result open, aborting with excluded if the first operand is not a Boolean (eval-and-nonbool, eval-or-nonbool), and a conditional expression requires its condition to be a Boolean, like the `if` statement (eval-if), aborting with excluded otherwise (eval-cond-nonbool).

A comprehension evaluates its body once for each binding produced by its qualifiers, and concatenates the results; a generator draws its bindings from the elements of a value, given by the auxiliary function iter (§A.4):

the elements of a list, tuple, string or range, and the keys of a dictionary, as in Python. A generator over a value other than a list, tuple, string, dictionary or range aborts with TypeError (eval-qual-generator-nonseq), as in Python, and a guard aborts with excluded if it evaluates to a non-Boolean (eval-qual-guard-nonbool). Dictionary expressions and comprehensions with keys other than strings abort with excluded (eval-dict-nonstr, eval-dict-comp-nonstr), as do subscripting and membership with such a key; Python raises `TypeError` only for keys it cannot hash, such as lists, but PurePy has no notion of hashing and cannot distinguish them from other keys. Subscripting is given by the auxiliary function getitem, and membership by in_Σ (§A.4, §A.4); both abort with excluded for a range, as does subscripting for a Boolean index, which Python treats as the integer 0 or 1. An attribute reference on a value other than a module or object aborts with excluded (eval-attr-nonobject). In this version of PurePy such values have no attributes, whereas in Python a string, for example, has methods; a reference rejected by Python with `AttributeError` also aborts with excluded, since PurePy does not keep track of which attributes Python provides.

$\vdash v : \theta$

val-lit
$$\frac{\Sigma \vdash \texttt{Literal}[\ell] <: \tau}{\vdash \ell : \tau}$$

val-lit-float
$$\frac{n \text{ a float literal} \qquad \Sigma \vdash \texttt{float} <: \tau}{\vdash n : \tau}$$

val-list
$$\frac{\vdash v_i : \sigma \quad (\forall i) \qquad \Sigma \vdash \texttt{list}[\sigma] <: \tau}{\vdash [\vec{v}] : \tau}$$

val-tuple
$$\frac{\vdash v_i : \sigma_i \quad (\forall i) \qquad \Sigma \vdash \texttt{tuple}[\vec{\sigma}] <: \tau}{\vdash (\vec{v}) : \tau}$$

val-dict
$$\frac{\vdash v_i : \sigma \quad (\forall i) \qquad \vec{w} \text{ distinct} \qquad \Sigma \vdash \texttt{dict}[\texttt{str}, \sigma] <: \tau}{\vdash \{\vec{w} : \vec{v}\} : \tau}$$

val-lambda
$$\frac{\vdash \rho : \Gamma \qquad \Gamma \otimes \{\vec{x} : \vec{\sigma}\} \vdash e \lhd \tau \qquad \Sigma \vdash \texttt{Callable}[[\vec{\sigma}], \tau] <: \tau'}{\vdash \mathsf{Lam}_\Gamma(\rho, \vec{x}, e) : \tau'}$$

val-def
$$\frac{\vdash \rho : \Gamma \qquad \mathrm{declares}(\vec{d}) = \vec{f} \qquad \Gamma \vdash \vec{d} : \mathsf{Assigns}(\Delta) \qquad \mathrm{fv}(\sigma_j) = \emptyset \quad (\forall j) \qquad \Sigma \vdash \Delta(f_i)[\vec{\sigma}] <: \tau}{\vdash \mathsf{Def}_\Gamma(\rho, \vec{d}, i) : \tau}$$

val-def-scheme
$$\frac{\vdash \rho : \Gamma \qquad \mathrm{declares}(\vec{d}) = \vec{f} \qquad \Gamma \vdash \vec{d} : \mathsf{Assigns}(\Delta) \qquad \Delta(f_i) = \forall \vec{\alpha}.\, \sigma \qquad \vec{\alpha} \text{ non-empty}}{\vdash \mathsf{Def}_\Gamma(\rho, \vec{d}, i) : \forall \vec{\alpha}.\, \sigma}$$

val-obj
$$\frac{\mathrm{fields}_\Sigma(C)[\vec{\sigma}] = \vec{x} : \vec{\sigma}' \qquad \vdash v_i : \sigma_i' \quad (\forall i) \qquad \Sigma \vdash C[\vec{\sigma}] <: \tau}{\vdash \mathsf{Obj}(C, \vec{x} : \vec{v}) : \tau}$$

val-prim
$$\frac{\mathcal{E}(\mathcal{A})(q)(x) = \mathsf{Prim}(f) \qquad \Sigma \vdash \mathcal{P}(q)(x) <: \tau}{\vdash \mathsf{Prim}(f) : \tau}$$

val-predef
$$\frac{}{\vdash \mathsf{Predef}(q.x) : \mathsf{Predef}}$$

val-mod
$$\frac{q \in \mathrm{dom}(\mathcal{M}) \cup \mathrm{dom}(\mathcal{P})}{\vdash \mathsf{Mod}(q) : \mathsf{Mod}(q)}$$

val-mod-loaded
$$\frac{\vdash \rho : \Gamma}{\vdash \mathsf{Mod}(q, \rho) : \mathsf{Mod}(q, \Gamma)}$$

val-class
$$\frac{C \in \mathrm{dom}(\Sigma)}{\vdash \mathsf{Type}(C) : \mathsf{Type}(C)}$$

val-type-alias
$$\frac{}{\vdash \mathsf{TypeAlias}_\Gamma(q, \vec{\chi}, i) : \mathsf{TypeAlias}_\Gamma(q, \vec{\chi}, i)}$$

val-unbound
$$\frac{\theta \in \{\mathsf{Unbound}, \mathsf{DU}(\tau)\}}{\vdash \mathsf{Unbound} : \theta}$$

val-pu
$$\frac{v = \mathsf{Unbound} \text{ or } \vdash v : \theta \qquad \theta \in \{\tau, \forall \vec{\alpha}.\, \tau\}}{\vdash v : \mathsf{PU}(\theta)}$$

(a) Value v is well-typed at context entry θ (`wf_val`)

$\vdash \rho : \Gamma$

env-nil
$$\frac{}{\vdash \emptyset : \emptyset}$$

env-cons
$$\frac{\vdash v : \theta \qquad \vdash \rho : \Gamma}{\vdash \{x : v\} \otimes \rho : \{x : \theta\} \otimes \Gamma}$$

(b) Environment ρ is well-typed at context Γ (`wf_env`)

Figure 4.3–1: Value and environment typing

$\rho, s \Rightarrow o$

eval-pass

$$\frac{}{\rho, \texttt{pass} \Rightarrow (\varepsilon, \mathsf{assigns}(\emptyset))}$$

eval-assign

$$\frac{\rho, e \Rightarrow r}{\rho, x = e \Rightarrow \mathsf{let}\ v \leftarrow r\ \mathsf{in}\ (\varepsilon, \mathsf{assigns}(\{x : v\}))}$$

eval-declare-assign

$$\frac{\rho, e \Rightarrow r}{\rho, x : \psi = e \Rightarrow \mathsf{let}\ v \leftarrow r\ \mathsf{in}\ (\varepsilon, \mathsf{assigns}(\{x : v\}))}$$

eval-declare

$$\frac{}{\rho, x : \psi \Rightarrow (\varepsilon, \mathsf{assigns}(\emptyset))}$$

eval-expr-stmt

$$\frac{\rho, e \Rightarrow r}{\rho, e \Rightarrow \mathsf{let}\ v \leftarrow r\ \mathsf{in}\ (\varepsilon, \mathsf{assigns}(\emptyset))}$$

eval-print

$$\frac{\rho, e \Rightarrow r}{\rho, \texttt{print}(e) \Rightarrow \mathsf{let}\ v \leftarrow r\ \mathsf{in}\ (v, \mathsf{assigns}(\emptyset))}$$

eval-return

$$\frac{\rho, e \Rightarrow r}{\rho, \texttt{return}\ e \Rightarrow \mathsf{let}\ v \leftarrow r\ \mathsf{in}\ (\varepsilon, \mathsf{returns}(v))}$$

eval-return-none

$$\frac{}{\rho, \texttt{return} \Rightarrow (\varepsilon, \mathsf{returns}(\texttt{None}))}$$

eval-defs

$$\frac{\mathrm{declares}(\vec{d}) = \vec{f} \qquad v_i = \mathsf{Def}_\Gamma(\rho, \vec{d}, i) \quad (\forall i)}{\rho, \vec{d} \Rightarrow (\varepsilon, \mathsf{assigns}(\{\vec{f} : \vec{v}\}))}$$

eval-assert

$$\frac{\rho, e \Rightarrow \mathsf{val}(\texttt{True})}{\rho, \texttt{assert}\ e \Rightarrow (\varepsilon, \mathsf{assigns}(\emptyset))}$$

eval-assert-msg

$$\frac{\rho, e \Rightarrow \mathsf{val}(\texttt{True})}{\rho, \texttt{assert}\ e, e' \Rightarrow (\varepsilon, \mathsf{assigns}(\emptyset))}$$

eval-assert-false

$$\frac{\rho, e \Rightarrow \mathsf{val}(\texttt{False})}{\rho, \texttt{assert}\ e \Rightarrow (\varepsilon, \mathsf{aborts}(\mathsf{AssertionError}))}$$

eval-assert-msg-false

$$\frac{\rho, e \Rightarrow \mathsf{val}(\texttt{False}) \qquad \rho, e' \Rightarrow r}{\rho, \texttt{assert}\ e, e' \Rightarrow \mathsf{let}\ w \leftarrow r\ \mathsf{in}\ (\varepsilon, \mathsf{aborts}(\mathsf{AssertionError}(w)))}$$

eval-if

$$\frac{\rho, e_j \Rightarrow \mathsf{val}(\texttt{False}) \quad (\forall j < i) \qquad \rho, e_i \Rightarrow \mathsf{val}(\texttt{True}) \qquad \rho, s_i \Rightarrow o}{\rho, \texttt{if}\ e_1 :\ s_1\ \texttt{elif}\ e_2 :\ s_2\ \ldots\ \texttt{elif}\ e_{n+1} :\ s_{n+1} \Rightarrow o}$$

eval-if-none

$$\frac{\rho, e_j \Rightarrow \mathsf{val}(\texttt{False}) \quad (\forall j \in \{1, \ldots, n+1\})}{\rho, \texttt{if}\ e_1 :\ s_1\ \texttt{elif}\ e_2 :\ s_2\ \ldots\ \texttt{elif}\ e_{n+1} :\ s_{n+1} \Rightarrow (\varepsilon, \mathsf{assigns}(\emptyset))}$$

eval-if-else

$$\frac{\rho, e_j \Rightarrow \mathsf{val}(\texttt{False}) \quad (\forall j < i) \qquad \rho, e_i \Rightarrow \mathsf{val}(\texttt{True}) \qquad \rho, s_i \Rightarrow o}{\rho, \texttt{if}\ e_1 :\ s_1\ \texttt{elif}\ e_2 :\ s_2\ \ldots\ \texttt{elif}\ e_{n+1} :\ s_{n+1}\ \texttt{else} :\ s_{n+2} \Rightarrow o}$$

eval-else

$$\frac{\rho, e_j \Rightarrow \mathsf{val}(\texttt{False}) \quad (\forall j \in \{1, \ldots, n+1\}) \qquad \rho, s_{n+2} \Rightarrow o}{\rho, \texttt{if}\ e_1 :\ s_1\ \texttt{elif}\ e_2 :\ s_2\ \ldots\ \texttt{elif}\ e_{n+1} :\ s_{n+1}\ \texttt{else} :\ s_{n+2} \Rightarrow o}$$

eval-match

$$\frac{\rho, e \Rightarrow \mathsf{val}(v) \qquad \mathrm{dispatch}(\rho, v, \vec{p}, \vec{s}) = (\rho', s') \qquad \rho \otimes \rho', s' \Rightarrow o}{\rho, \texttt{match}\ e\texttt{:}\ \texttt{case}\ p_1\texttt{:}\ s_1\ \ldots\ \texttt{case}\ p_{n+1}\texttt{:}\ s_{n+1} \Rightarrow (\varepsilon, \mathsf{assigns}(\rho')) \otimes o}$$

eval-seq

$$\frac{\rho, s \Rightarrow (\vec{u}, \mathsf{assigns}(\rho')) \qquad \rho \otimes \rho', s' \Rightarrow o}{\rho, s\ s' \Rightarrow (\vec{u}, \mathsf{assigns}(\rho')) \otimes o}$$

eval-seq-return

$$\frac{\rho, s \Rightarrow (\vec{u}, \mathsf{returns}(v))}{\rho, s\ s' \Rightarrow (\vec{u}, \mathsf{returns}(v))}$$

Figure 4.3–2: Statement s evaluates to outcome o under environment $\rho : \Gamma$ (`eval_stmt`)

$\rho, s \Rightarrow o$ (continued)

eval-assert-fail
$$\frac{\rho, e \Rightarrow \mathsf{aborts}(\kappa)}{\rho, \texttt{assert}\ e \Rightarrow (\varepsilon, \mathsf{aborts}(\kappa))}$$

eval-assert-msg-fail
$$\frac{\rho, e \Rightarrow \mathsf{aborts}(\kappa)}{\rho, \texttt{assert}\ e, e' \Rightarrow (\varepsilon, \mathsf{aborts}(\kappa))}$$

eval-assert-nonbool
$$\frac{\rho, e \Rightarrow \mathsf{val}(v) \qquad v \text{ not a Boolean}}{\rho, \texttt{assert}\ e \Rightarrow (\varepsilon, \mathsf{aborts}(\mathsf{excluded}))}$$

eval-assert-msg-nonbool
$$\frac{\rho, e \Rightarrow \mathsf{val}(v) \qquad v \text{ not a Boolean}}{\rho, \texttt{assert}\ e, e' \Rightarrow (\varepsilon, \mathsf{aborts}(\mathsf{excluded}))}$$

eval-assert-msg-nonstr
$$\frac{\rho, e \Rightarrow \mathsf{val}(\texttt{False}) \qquad \rho, e' \Rightarrow \mathsf{val}(v) \qquad v \text{ not a string}}{\rho, \texttt{assert}\ e, e' \Rightarrow (\varepsilon, \mathsf{aborts}(\mathsf{excluded}))}$$

eval-if-fail
$$\frac{\rho, e_j \Rightarrow \mathsf{val}(\texttt{False}) \quad (\forall j < i) \qquad \rho, e_i \Rightarrow \mathsf{aborts}(\kappa)}{\rho, \texttt{if}\ e_1 : s_1\ \texttt{elif}\ e_2 : s_2\ \ldots\ \texttt{elif}\ e_{n+1} : s_{n+1} \Rightarrow (\varepsilon, \mathsf{aborts}(\kappa))}$$

eval-if-else-fail
$$\frac{\rho, e_j \Rightarrow \mathsf{val}(\texttt{False}) \quad (\forall j < i) \qquad \rho, e_i \Rightarrow \mathsf{aborts}(\kappa)}{\rho, \texttt{if}\ e_1 : s_1\ \texttt{elif}\ e_2 : s_2\ \ldots\ \texttt{elif}\ e_{n+1} : s_{n+1}\ \texttt{else} : s_{n+2} \Rightarrow (\varepsilon, \mathsf{aborts}(\kappa))}$$

eval-if-nonbool
$$\frac{\rho, e_j \Rightarrow \mathsf{val}(\texttt{False}) \quad (\forall j < i) \qquad \rho, e_i \Rightarrow \mathsf{val}(v) \qquad v \text{ not a Boolean}}{\rho, \texttt{if}\ e_1 : s_1\ \texttt{elif}\ e_2 : s_2\ \ldots\ \texttt{elif}\ e_{n+1} : s_{n+1} \Rightarrow (\varepsilon, \mathsf{aborts}(\mathsf{excluded}))}$$

eval-if-else-nonbool
$$\frac{\rho, e_j \Rightarrow \mathsf{val}(\texttt{False}) \quad (\forall j < i) \qquad \rho, e_i \Rightarrow \mathsf{val}(v) \qquad v \text{ not a Boolean}}{\rho, \texttt{if}\ e_1 : s_1\ \texttt{elif}\ e_2 : s_2\ \ldots\ \texttt{elif}\ e_{n+1} : s_{n+1}\ \texttt{else} : s_{n+2} \Rightarrow (\varepsilon, \mathsf{aborts}(\mathsf{excluded}))}$$

eval-match-fail
$$\frac{\rho, e \Rightarrow \mathsf{aborts}(\kappa)}{\rho, \texttt{match}\ e\texttt{:}\ \texttt{case}\ p_1\texttt{:}\ s_1\ \ldots\ \texttt{case}\ p_{n+1}\texttt{:}\ s_{n+1} \Rightarrow (\varepsilon, \mathsf{aborts}(\kappa))}$$

eval-match-abort
$$\frac{\rho, e \Rightarrow \mathsf{val}(v) \qquad \mathrm{dispatch}(\rho, v, \vec{p}, \vec{s}) = \kappa}{\rho, \texttt{match}\ e\texttt{:}\ \texttt{case}\ p_1\texttt{:}\ s_1\ \ldots\ \texttt{case}\ p_{n+1}\texttt{:}\ s_{n+1} \Rightarrow (\varepsilon, \mathsf{aborts}(\kappa))}$$

eval-seq-fail
$$\frac{\rho, s \Rightarrow (\vec{u}, \mathsf{aborts}(\kappa))}{\rho, s\ s' \Rightarrow (\vec{u}, \mathsf{aborts}(\kappa))}$$

Figure 4.3–3: Statement evaluation, continued (propagation rules and aborting cases) (`eval_stmt`)

$q, \rho, t \Rightarrow o$

eval-top-stmt
$$\frac{\rho, s \Rightarrow o}{q, \rho, s \Rightarrow o}$$

eval-top-seq
$$\frac{q, \rho, t \Rightarrow (\vec{u}, \mathsf{assigns}(\rho')) \qquad q, \rho \otimes \rho', t' \Rightarrow o}{q, \rho, t\ t' \Rightarrow (\vec{u}, \mathsf{assigns}(\rho')) \otimes o}$$

eval-top-seq-fail
$$\frac{q, \rho, t \Rightarrow (\vec{u}, \mathsf{aborts}(\kappa))}{q, \rho, t\ t' \Rightarrow (\vec{u}, \mathsf{aborts}(\kappa))}$$

eval-types
$$\frac{\rho' = \{\mathrm{declares}(\vec{\chi})\texttt{:}\ \text{type-entry}_\Gamma(q, \vec{\chi})\}}{q, \rho, \vec{\chi} \Rightarrow (\varepsilon, \mathsf{assigns}(\rho'))}$$

Figure 4.3–4: Top-level statement t of module q evaluates to outcome o under environment $\rho : \Gamma$ (`eval_top`)

$\rho, e \Rightarrow r$

eval-var
$$\frac{x : v \in \rho \qquad v \neq \mathsf{Unbound}}{\rho, x \Rightarrow \mathsf{val}(v)}$$

eval-literal
$$\frac{}{\rho, \ell \Rightarrow \mathsf{val}(\ell)}$$

eval-lambda
$$\frac{}{\rho, \texttt{lambda}\ \vec{x} : \ e \Rightarrow \mathsf{val}(\mathsf{Lam}_\Gamma(\rho, \vec{x}, e))}$$

eval-unop
$$\frac{\rho, e \Rightarrow r}{\rho, \ominus e \Rightarrow \mathsf{let}\ v \leftarrow r\ \mathsf{in}\ \hat{\ominus}(v)}$$

eval-binop
$$\frac{\rho, e \Rightarrow \mathsf{val}(v) \qquad \rho, e' \Rightarrow r}{\rho, e \odot e' \Rightarrow \mathsf{let}\ v' \leftarrow r\ \mathsf{in}\ \hat{\ominus}(v, v')}$$

eval-list
$$\frac{\rho, \vec{e} \Rightarrow \mathsf{val}(\vec{v})}{\rho, [\vec{e}] \Rightarrow \mathsf{val}([\vec{v}])}$$

eval-tuple
$$\frac{\rho, \vec{e} \Rightarrow \mathsf{val}(\vec{v})}{\rho, (\vec{e}) \Rightarrow \mathsf{val}((\vec{v}))}$$

eval-dict
$$\frac{\rho, e_1 \cdot e'_1 \cdot \ldots \cdot e_n \cdot e'_n \Rightarrow \mathsf{val}(w_1 \cdot v_1 \cdot \ldots \cdot w_n \cdot v_n)}{\rho, \{e_1 : e'_1, \ldots, e_n : e'_n\} \Rightarrow \mathsf{val}(\{\mathrm{update}(\varepsilon, w_1 : v_1 \cdot \ldots \cdot w_n : v_n)\})}$$

eval-list-comp
$$\frac{\rho, \vec{g} \Rightarrow \mathsf{val}(\vec{\rho}) \qquad \rho_i, e \Rightarrow \mathsf{val}(v_i) \quad (\forall i)}{\rho, [\ e\ \vec{g}\ ] \Rightarrow \mathsf{val}([\vec{v}])}$$

eval-dict-comp
$$\frac{\rho, \vec{g} \Rightarrow \mathsf{val}(\vec{\rho}) \qquad \rho_i, e \Rightarrow \mathsf{val}(w_i) \quad (\forall i) \qquad \rho_i, e' \Rightarrow \mathsf{val}(v_i) \quad (\forall i)}{\rho, \{e : e'\ \vec{g}\} \Rightarrow \mathsf{val}(\{\mathrm{update}(\varepsilon, \vec{w} : \vec{v})\})}$$

eval-attr-module
$$\frac{\rho, e \Rightarrow \mathsf{val}(\mathsf{Mod}(q', \rho')) \qquad x : v \in \rho'}{\rho, e.x \Rightarrow \mathsf{val}(v)}$$

eval-subscript
$$\frac{\rho, e \Rightarrow \mathsf{val}(v) \qquad v \text{ not a class} \qquad \rho, e' \Rightarrow r}{\rho, e[e'] \Rightarrow \mathsf{let}\ v' \leftarrow r\ \mathsf{in}\ \mathrm{getitem}(v, v')}$$

eval-class-type-args
$$\frac{\rho, e \Rightarrow \mathsf{val}(\mathsf{Type}(C))}{\rho, e[\vec{\psi}] \Rightarrow \mathsf{val}(\mathsf{Type}(C))}$$

eval-attr-object
$$\frac{\rho, e \Rightarrow \mathsf{val}(\mathsf{Obj}(C, \vec{x} : \vec{v})) \qquad x = x_i}{\rho, e.x \Rightarrow \mathsf{val}(v_i)}$$

eval-constr
$$\frac{\rho, e \Rightarrow \mathsf{val}(\mathsf{Type}(C)) \qquad \rho, \vec{e} +\!\!+ \vec{e'} \Rightarrow \mathsf{val}(\vec{v} +\!\!+ \vec{v'}) \qquad \mathrm{fields}_\Sigma(C) = \forall \vec{\alpha}.\, \vec{x} : \vec{\sigma} \qquad \text{field-map}_\Sigma(C, (\vec{v}, \vec{y}{=}\vec{v'})) = \{\vec{x} : \vec{u}\}}{\rho, e(\vec{e}, \vec{y}{=}\vec{e'}) \Rightarrow \mathsf{val}(\mathsf{Obj}(C, \vec{x} : \vec{u}))}$$

eval-call-lambda
$$\frac{\rho, e \Rightarrow \mathsf{val}(\mathsf{Lam}_\Delta(\rho', \vec{x}, e')) \qquad \rho, \vec{e} \Rightarrow \mathsf{val}(\vec{u}) \qquad \rho' \otimes \{\vec{x} : \vec{u}\}, e' \Rightarrow r}{\rho, e(\vec{e}) \Rightarrow r}$$

eval-call-def
$$\frac{\begin{array}{c}\rho, e \Rightarrow \mathsf{val}(\mathsf{Def}_\Delta(\rho', \vec{d}, j)) \qquad d_j = \texttt{def}\ f[\vec{\alpha}](\vec{x} : \vec{\psi})\ \texttt{->}\ \psi : \ s \\ v_i = \mathsf{Def}_\Delta(\rho', \vec{d}, i) \quad (\forall i) \qquad \rho, \vec{e} \Rightarrow \mathsf{val}(\vec{u}) \qquad \rho' \otimes \{\mathrm{declares}(\vec{d}) : \vec{v}\} \otimes \{\vec{x} : \vec{u}\} \otimes \mathrm{scope}(\vec{x}, s), s \Rightarrow o\end{array}}{\rho, e(\vec{e}) \Rightarrow \mathrm{result}(o)}$$

eval-call-prim
$$\frac{\rho, e \Rightarrow \mathsf{val}(\mathsf{Prim}(f)) \qquad \rho, \vec{e} \Rightarrow \mathsf{val}(\vec{u})}{\rho, e(\vec{e}) \Rightarrow f(\vec{u})}$$

eval-and-false
$$\frac{\rho, e \Rightarrow \mathsf{val}(\texttt{False})}{\rho, e\ \texttt{and}\ e' \Rightarrow \mathsf{val}(\texttt{False})}$$

eval-and-true
$$\frac{\rho, e \Rightarrow \mathsf{val}(\texttt{True}) \qquad \rho, e' \Rightarrow r}{\rho, e\ \texttt{and}\ e' \Rightarrow r}$$

eval-or-true
$$\frac{\rho, e \Rightarrow \mathsf{val}(\texttt{True})}{\rho, e\ \texttt{or}\ e' \Rightarrow \mathsf{val}(\texttt{True})}$$

eval-or-false
$$\frac{\rho, e \Rightarrow \mathsf{val}(\texttt{False}) \qquad \rho, e' \Rightarrow r}{\rho, e\ \texttt{or}\ e' \Rightarrow r}$$

eval-cond-true
$$\frac{\rho, e \Rightarrow \mathsf{val}(\texttt{True}) \qquad \rho, e_1 \Rightarrow r}{\rho, e_1\ \texttt{if}\ e\ \texttt{else}\ e_2 \Rightarrow r}$$

eval-cond-false
$$\frac{\rho, e \Rightarrow \mathsf{val}(\texttt{False}) \qquad \rho, e_2 \Rightarrow r}{\rho, e_1\ \texttt{if}\ e\ \texttt{else}\ e_2 \Rightarrow r}$$

Figure 4.3–5: Expression e evaluates to result r under environment $\rho : \Gamma$ (`eval_expr`)

$\rho, e \Rightarrow r$ (continued)

eval-binop-fail
$$\frac{\rho, e \Rightarrow \text{aborts}(\kappa)}{\rho, e \odot e' \Rightarrow \text{aborts}(\kappa)}$$

eval-and-fail
$$\frac{\rho, e \Rightarrow \text{aborts}(\kappa)}{\rho, e \texttt{ and } e' \Rightarrow \text{aborts}(\kappa)}$$

eval-or-fail
$$\frac{\rho, e \Rightarrow \text{aborts}(\kappa)}{\rho, e \texttt{ or } e' \Rightarrow \text{aborts}(\kappa)}$$

eval-cond-fail
$$\frac{\rho, e \Rightarrow \text{aborts}(\kappa)}{\rho, e_1 \texttt{ if } e \texttt{ else } e_2 \Rightarrow \text{aborts}(\kappa)}$$

eval-attr-fail
$$\frac{\rho, e \Rightarrow \text{aborts}(\kappa)}{\rho, e.x \Rightarrow \text{aborts}(\kappa)}$$

eval-subscript-fail
$$\frac{\rho, e \Rightarrow \text{aborts}(\kappa)}{\rho, e[e'] \Rightarrow \text{aborts}(\kappa)}$$

eval-call-fail
$$\frac{\rho, e \Rightarrow \text{aborts}(\kappa)}{\rho, e(\vec{e}, \vec{y}{=}\vec{e'}) \Rightarrow \text{aborts}(\kappa)}$$

eval-call-fail-args
$$\frac{\rho, e \Rightarrow \text{val}(v) \qquad \rho, \vec{e} \Rightarrow \text{aborts}(\kappa)}{\rho, e(\vec{e}) \Rightarrow \text{aborts}(\kappa)}$$

eval-list-fail
$$\frac{\rho, \vec{e} \Rightarrow \text{aborts}(\kappa)}{\rho, [\vec{e}] \Rightarrow \text{aborts}(\kappa)}$$

eval-tuple-fail
$$\frac{\rho, \vec{e} \Rightarrow \text{aborts}(\kappa)}{\rho, (\vec{e}) \Rightarrow \text{aborts}(\kappa)}$$

eval-dict-fail
$$\frac{\rho, e_1 \cdot e'_1 \cdot \ldots \cdot e_n \cdot e'_n \Rightarrow \text{aborts}(\kappa)}{\rho, \{e_1 : e'_1, \ldots, e_n : e'_n\} \Rightarrow \text{aborts}(\kappa)}$$

eval-list-comp-fail-quals
$$\frac{\rho, \vec{g} \Rightarrow \text{aborts}(\kappa)}{\rho, [\ e\ \vec{g}] \Rightarrow \text{aborts}(\kappa)}$$

eval-list-comp-fail
$$\frac{\rho, \vec{g} \Rightarrow \text{val}(\vec{\rho}) \qquad \rho_i, e \Rightarrow \text{val}(v_i) \quad (\forall i < j) \qquad \rho_j, e \Rightarrow \text{aborts}(\kappa)}{\rho, [\ e\ \vec{g}] \Rightarrow \text{aborts}(\kappa)}$$

eval-dict-comp-fail-quals
$$\frac{\rho, \vec{g} \Rightarrow \text{aborts}(\kappa)}{\rho, \{e : e'\ \vec{g}\} \Rightarrow \text{aborts}(\kappa)}$$

eval-dict-comp-fail-key
$$\frac{\rho, \vec{g} \Rightarrow \text{val}(\vec{\rho}) \qquad \rho_i, e \Rightarrow \text{val}(w_i) \quad \rho_i, e' \Rightarrow \text{val}(v_i) \quad (\forall i < j) \qquad \rho_j, e \Rightarrow \text{aborts}(\kappa)}{\rho, \{e : e'\ \vec{g}\} \Rightarrow \text{aborts}(\kappa)}$$

eval-dict-comp-fail-value
$$\frac{\rho, \vec{g} \Rightarrow \text{val}(\vec{\rho}) \qquad \rho_i, e \Rightarrow \text{val}(w_i) \quad \rho_i, e' \Rightarrow \text{val}(v_i) \quad (\forall i < j) \qquad \rho_j, e \Rightarrow \text{val}(w_j) \qquad \rho_j, e' \Rightarrow \text{aborts}(\kappa)}{\rho, \{e : e'\ \vec{g}\} \Rightarrow \text{aborts}(\kappa)}$$

eval-constr-fail
$$\frac{\rho, e \Rightarrow \text{val}(\text{Type}(C)) \qquad \rho, \vec{e} +\!\!+ \vec{e'} \Rightarrow \text{aborts}(\kappa)}{\rho, e(\vec{e}, \vec{y}{=}\vec{e'}) \Rightarrow \text{aborts}(\kappa)}$$

Figure 4.3–6: Expression evaluation, continued (propagation rules) (`eval_expr`)

$\rho, e \Rightarrow r$ (continued)

eval-var-unknown
$$\frac{x \notin \mathrm{dom}(\rho)}{\rho, x \Rightarrow \mathsf{aborts}(\mathsf{NameError})}$$

eval-var-unbound
$$\frac{x : \mathsf{Unbound} \in \rho}{\rho, x \Rightarrow \mathsf{aborts}(\mathsf{NameError})}$$

eval-attr-object-missing
$$\frac{\rho, e \Rightarrow \mathsf{val}(\mathsf{Obj}(C, \vec{x} : \vec{v})) \qquad x \notin \{\vec{x}\}}{\rho, e.x \Rightarrow \mathsf{aborts}(\mathsf{AttributeError})}$$

eval-attr-module-missing
$$\frac{\rho, e \Rightarrow \mathsf{val}(\mathsf{Mod}(q', \rho')) \qquad x \notin \mathrm{dom}(\rho')}{\rho, e.x \Rightarrow \mathsf{aborts}(\mathsf{AttributeError})}$$

eval-call-lambda-arity
$$\frac{\rho, e \Rightarrow \mathsf{val}(\mathsf{Lam}_\Delta(\rho', \vec{x}, e')) \qquad \rho, \vec{e} \Rightarrow \mathsf{val}(\vec{u}) \qquad |\vec{x}| \neq |\vec{u}|}{\rho, e(\vec{e}) \Rightarrow \mathsf{aborts}(\mathsf{TypeError})}$$

eval-call-def-arity
$$\frac{\rho, e \Rightarrow \mathsf{val}(\mathsf{Def}_\Delta(\rho', \vec{d}, j)) \qquad d_j = \texttt{def}\ f[\vec{\alpha}](\vec{x} : \vec{\psi})\ \texttt{->}\ \psi : s \qquad \rho, \vec{e} \Rightarrow \mathsf{val}(\vec{u}) \qquad |\vec{x}| \neq |\vec{u}|}{\rho, e(\vec{e}) \Rightarrow \mathsf{aborts}(\mathsf{TypeError})}$$

eval-constr-arity
$$\frac{\rho, e \Rightarrow \mathsf{val}(\mathsf{Type}(C)) \qquad \rho, \vec{e} +\!\!+ \vec{e}' \Rightarrow \mathsf{val}(\vec{v} +\!\!+ \vec{v}') \qquad \text{field-map}_\Sigma(C, (\vec{v}, \vec{y}{=}\vec{v}'))\ \text{undefined}}{\rho, e(\vec{e}, \vec{y}{=}\vec{e}') \Rightarrow \mathsf{aborts}(\mathsf{TypeError})}$$

eval-call-nonfun
$$\frac{\rho, e \Rightarrow \mathsf{val}(v) \qquad \rho, \vec{e} \Rightarrow \mathsf{val}(\vec{u}) \qquad v\ \text{neither closure, primitive nor class}}{\rho, e(\vec{e}) \Rightarrow \mathsf{aborts}(\mathsf{TypeError})}$$

eval-and-nonbool
$$\frac{\rho, e \Rightarrow \mathsf{val}(v) \qquad v\ \text{not a Boolean}}{\rho, e\ \texttt{and}\ e' \Rightarrow \mathsf{aborts}(\mathsf{excluded})}$$

eval-or-nonbool
$$\frac{\rho, e \Rightarrow \mathsf{val}(v) \qquad v\ \text{not a Boolean}}{\rho, e\ \texttt{or}\ e' \Rightarrow \mathsf{aborts}(\mathsf{excluded})}$$

eval-cond-nonbool
$$\frac{\rho, e \Rightarrow \mathsf{val}(v) \qquad v\ \text{not a Boolean}}{\rho, e_1\ \texttt{if}\ e\ \texttt{else}\ e_2 \Rightarrow \mathsf{aborts}(\mathsf{excluded})}$$

eval-attr-nonobject
$$\frac{\rho, e \Rightarrow \mathsf{val}(v) \qquad v\ \text{neither loaded module nor object}}{\rho, e.x \Rightarrow \mathsf{aborts}(\mathsf{excluded})}$$

eval-dict-nonstr
$$\frac{\rho, e_1 \cdot e_1' \cdot \ldots \cdot e_n \cdot e_n' \Rightarrow \mathsf{val}(v_1 \cdot v_1' \cdot \ldots \cdot v_n \cdot v_n') \qquad v_j\ \text{not a string}}{\rho, \{e_1 : e_1', \ldots, e_n : e_n'\} \Rightarrow \mathsf{aborts}(\mathsf{excluded})}$$

eval-dict-comp-nonstr
$$\frac{\rho, \vec{g} \Rightarrow \mathsf{val}(\vec{\rho}) \qquad \rho_i, e \Rightarrow \mathsf{val}(w_i) \qquad \rho_i, e' \Rightarrow \mathsf{val}(v_i) \qquad (\forall i < j) \qquad \rho_j, e \Rightarrow \mathsf{val}(v) \qquad v\ \text{not a string}}{\rho, \{e : e'\ \vec{g}\} \Rightarrow \mathsf{aborts}(\mathsf{excluded})}$$

Figure 4.3–7: Expression evaluation, continued (aborting cases) (`eval_expr`)

$\rho, \vec{e} \Rightarrow r$

eval-exprs-nil
$$\frac{}{\rho, \varepsilon \Rightarrow \mathsf{val}(\varepsilon)}$$

eval-exprs-cons
$$\frac{\rho, e \Rightarrow \mathsf{val}(v) \qquad \rho, \vec{e} \Rightarrow r}{\rho, e \cdot \vec{e} \Rightarrow \mathsf{let}\ \vec{v} \leftarrow r\ \mathsf{in}\ \mathsf{val}(v \cdot \vec{v})}$$

eval-exprs-fail
$$\frac{\rho, e \Rightarrow \mathsf{aborts}(\kappa)}{\rho, e \cdot \vec{e} \Rightarrow \mathsf{aborts}(\kappa)}$$

Figure 4.3–8: Expression sequence $\vec{e}$ evaluates left to right to result r under environment $\rho : \Gamma$ (`eval_exprs`)

$\rho, \vec{g} \Rightarrow r$

eval-quals-nil

$$\frac{}{\rho, \varepsilon \Rightarrow \mathsf{val}(\rho)}$$

eval-qual-guard-true

$$\frac{\rho, e \Rightarrow \mathsf{val}(\texttt{True}) \qquad \rho, \vec{g} \Rightarrow r}{\rho, \texttt{if}\ e \cdot \vec{g} \Rightarrow r}$$

eval-qual-guard-false

$$\frac{\rho, e \Rightarrow \mathsf{val}(\texttt{False})}{\rho, \texttt{if}\ e \cdot \vec{g} \Rightarrow \mathsf{val}(\varepsilon)}$$

eval-qual-guard-fail

$$\frac{\rho, e \Rightarrow \mathsf{aborts}(\kappa)}{\rho, \texttt{if}\ e \cdot \vec{g} \Rightarrow \mathsf{aborts}(\kappa)}$$

eval-qual-guard-nonbool

$$\frac{\rho, e \Rightarrow \mathsf{val}(v) \qquad v \text{ not a Boolean}}{\rho, \texttt{if}\ e \cdot \vec{g} \Rightarrow \mathsf{aborts}(\mathsf{excluded})}$$

eval-qual-generator

$$\frac{\rho, e \Rightarrow \mathsf{val}(v) \qquad \mathrm{iter}(v) = v_1 \cdot \ldots \cdot v_n \qquad \rho \otimes \{x : v_i\}, \vec{g} \Rightarrow \mathsf{val}(\vec{\rho}_i) \quad (\forall i)}{\rho, \texttt{for}\ x\ \texttt{in}\ e \cdot \vec{g} \Rightarrow \mathsf{val}(\vec{\rho}_1 +\!\!+ \ldots +\!\!+ \vec{\rho}_n)}$$

eval-qual-generator-fail

$$\frac{\rho, e \Rightarrow \mathsf{aborts}(\kappa)}{\rho, \texttt{for}\ x\ \texttt{in}\ e \cdot \vec{g} \Rightarrow \mathsf{aborts}(\kappa)}$$

eval-qual-generator-nonseq

$$\frac{\rho, e \Rightarrow \mathsf{val}(v) \qquad \mathrm{iter}(v) \text{ undefined}}{\rho, \texttt{for}\ x\ \texttt{in}\ e \cdot \vec{g} \Rightarrow \mathsf{aborts}(\mathsf{TypeError})}$$

eval-qual-generator-fail-body

$$\frac{\rho, e \Rightarrow \mathsf{val}(v) \qquad \mathrm{iter}(v) = v_1 \cdot \ldots \cdot v_n \qquad \rho \otimes \{x : v_i\}, \vec{g} \Rightarrow \mathsf{val}(\vec{\rho}_i) \quad (\forall i < j) \qquad \rho \otimes \{x : v_j\}, \vec{g} \Rightarrow \mathsf{aborts}(\kappa)}{\rho, \texttt{for}\ x\ \texttt{in}\ e \cdot \vec{g} \Rightarrow \mathsf{aborts}(\kappa)}$$

Figure 4.3–9: Comprehension qualifiers $\vec{g}$ evaluate to result r under environment $\rho : \Gamma$ (`eval_quals`)

4.4 Pattern matching

A `match` statement takes the first case whose pattern matches the value, running its body under the bindings produced by the pattern, and does nothing if no case matches; the auxiliary function dispatch (§A.4) selects the matching case. Whether a pattern matches is determined by the auxiliary function $\mathrm{match}_\Sigma(\rho, p, v)$ (§A.4). A *match result* m is $\mathsf{match}(\rho')$, a match, where ρ' binds the variables of the pattern to the values that they matched; $\mathsf{no\text{-}match}$, a failure to match; or a termination kind κ, if matching aborts, such as undef for a numeric literal pattern if the equality of the literal and the value is undefined. When matching aborts with κ before any pattern matches, dispatch gives κ, and the run aborts with κ (eval-match-abort). Matching a list pattern against a tuple, a tuple pattern against a list, or either against a range aborts with $\mathsf{excluded}$ (§A.4); a constructor pattern against an object aborts with $\mathsf{TypeError}$ or $\mathsf{excluded}$ when the sub-patterns do not cover the fields of the class (§A.4). A constructor pattern whose head does not denote a class aborts with $\mathsf{NameError}$, $\mathsf{AttributeError}$ or $\mathsf{TypeError}$, as in Python, or with $\mathsf{excluded}$ for a predefined class such as `int`, which Python matches by an instance test (§A.4).

Definition 4.3 (Composition of match results)**.** The match results of sub-patterns compose with $\otimes$, matching sequentially, so that the first failure or abort determines the result:

$$\mathsf{match}(\rho) \otimes \mathsf{match}(\rho') = \mathsf{match}((\rho \uplus \rho'))$$
$$\mathsf{match}(\rho) \otimes \mathsf{no\text{-}match} = \mathsf{no\text{-}match} \otimes m = \mathsf{no\text{-}match}$$
$$\mathsf{match}(\rho) \otimes \kappa = \kappa \otimes m = \kappa$$

with $\mathsf{match}(\emptyset)$ as left and right unit.

A constructor pattern matches any object whose class has the pattern's class among its ancestors, and its sub-patterns bind the fields of the pattern's class. Since an object records its class by name, the class test of match_Σ is a comparison of fully-qualified names.

4.5 Modules and programs

Module and program evaluation are defined in Figure 4.5–1. A program runs each module body once, after its imports have loaded, in *load order* (§A.3). Load order is fixed by the imports alone, since imports precede statements and so cannot depend on the outcome of a statement. Module evaluation $q, \mathcal{L} \Rightarrow_{\mathsf{mod}} o$ runs the body of module q given the *loaded module environments* $\mathcal{L}$, a finite map from module names to the environments of the modules loaded so far. The body runs in the environment corresponding to the context in which the body was checked (eval-module). Import bindings combine by context extension (Definition 3.5), which is extended to environments by the same equations, but with values in place of context entries and environments in place of contexts.

Evaluation of a sequence of module bodies extends $\mathcal{L}$ with the environment of each module in turn (eval-modules-cons). Following the module rule (Figure 3.5–3a), the environment of a module q consists of a stub for each submodule of q, overridden by the environment assigned by the body of q. The program rules start from the environments $\mathcal{E}(\mathcal{A})$ of the predefined modules, given the command-line arguments $\mathcal{A}$ (eval-program and eval-program-abort).

A program evaluates to a *program result* $(\vec{v}, \kappa)$, pairing its output $\vec{v}$ with a termination kind κ (Figure 4.1–1). The output is the concatenation of the values printed by the module bodies, and the termination kind is $\mathsf{SystemExit}(0)$ if every module body completes normally, and otherwise the kind of the first module body to abort, after which nothing more is printed. $\mathsf{SystemExit}(n)$ is a run that exited with status n.

Definition 4.4 (Conformance)**.** An implementation *conforms* to the semantics if, for every well-formed program $\mathcal{M}$, arguments $\mathcal{A}$ and program result $(\vec{v}, \kappa)$ with $\mathcal{M}, \mathcal{A} \Rightarrow \vec{v}, \kappa$ and $\kappa \neq \mathsf{undef}$, a run of the implementation on $\mathcal{M}$ with arguments $\mathcal{A}$ writes the values $\vec{v}$ in order, each as by Python's `str` followed by a newline, and reports termination kind κ.

The form in which an implementation reports the termination kind is not prescribed. An implementation also conforms if it meets this definition with the result of `**` on floats being the nearest float below or above the exact power, not necessarily the nearer of the two. Python implementations conform, since they compute `**` on floats with functions such as `pow` in C and `Math.pow` in Java, which meet this requirement.

$q, \mathcal{L} \Rightarrow_{\text{mod}} o$

$$\text{eval-module}\quad \frac{\mathcal{M}(q) = \vec{\imath}\ t \qquad q, \mathcal{L}(\texttt{builtins}) \otimes \text{import-env}_{\mathcal{L}}(\vec{\imath}) \otimes \text{scope}(\text{declares}(\vec{\imath}), t), t \Rightarrow o}{q, \mathcal{L} \Rightarrow_{\text{mod}} o}$$

(a) Body of module q evaluates to outcome o given loaded module environments $\mathcal{L}$ (`eval_mod`)

$\vec{q}, \mathcal{L} \Rightarrow_{\text{mod}} o$

$$\text{eval-modules-nil}\quad \frac{}{\varepsilon, \mathcal{L} \Rightarrow_{\text{mod}} (\varepsilon, \text{assigns}(\emptyset))}$$

$$\text{eval-modules-cons}\quad \frac{q, \mathcal{L} \Rightarrow_{\text{mod}} (\vec{v}, \text{assigns}(\rho)) \qquad \vec{q}, \mathcal{L} \otimes \{q : \text{submods}(q) \otimes \rho\} \Rightarrow_{\text{mod}} o}{q \cdot \vec{q}, \mathcal{L} \Rightarrow_{\text{mod}} (\vec{v}, \text{assigns}(\emptyset)) \otimes o}$$

$$\text{eval-modules-fail}\quad \frac{q, \mathcal{L} \Rightarrow_{\text{mod}} (\vec{v}, \text{aborts}(\kappa))}{q \cdot \vec{q}, \mathcal{L} \Rightarrow_{\text{mod}} (\vec{v}, \text{aborts}(\kappa))}$$

(b) Bodies of modules $\vec{q}$ evaluate to outcome o given loaded module environments $\mathcal{L}$ (`eval_mods`)

$\mathcal{M}, \mathcal{A} \Rightarrow \vec{v}, \kappa$

$$\text{eval-program}\quad \frac{\text{loads}(\varepsilon, \texttt{__main__}), \mathcal{E}(\mathcal{A}) \Rightarrow_{\text{mod}} (\vec{v}, \text{assigns}(\rho))}{\mathcal{M}, \mathcal{A} \Rightarrow \vec{v}, \text{SystemExit}(0)}$$

$$\text{eval-program-abort}\quad \frac{\text{loads}(\varepsilon, \texttt{__main__}), \mathcal{E}(\mathcal{A}) \Rightarrow_{\text{mod}} (\vec{v}, \text{aborts}(\kappa))}{\mathcal{M}, \mathcal{A} \Rightarrow \vec{v}, \kappa}$$

(c) Program $\mathcal{M}$ run with arguments $\mathcal{A}$ prints $\vec{v}$ and ends with termination kind κ (`eval_program`)

Figure 4.5–1: Module and program evaluation

4.6 Type soundness

The termination kinds TypeError, AttributeError, NameError and excluded record operations which the type system rules out (§B.3).

Theorem 4.1 (Type soundness; `type_soundness`). *If* $\vdash \mathcal{M}$ *and* $\mathcal{M}, \mathcal{A} \Rightarrow \vec{v}, \kappa$ *then* κ *is not* TypeError, AttributeError, NameError *or* excluded.

Annex A

Auxiliary functions (normative)

This annex is normative. It defines the auxiliary functions used by the well-formedness rules (Chapter 3) and the operational semantics (Chapter 4). When more than one case applies, the first defines the value, so a function with overlapping cases is still determined: two equivalent types join to the second of them, for instance.

A.1 Functions on syntax

The syntactic analyses introduced in Chapter 3 are defined by their principal cases; the remaining cases are stated with each.

assigns(t) — *set of variables*

Variables declared or assigned anywhere in t, not descending into nested functions, which have their own scope. It is empty on the statements not shown and is the union over the branches of a conditional, and on a module body $\vec{\imath}\ t$ acts on t.

$$\begin{aligned}
\text{assigns}(x\texttt{:}\ \psi) = \text{assigns}(x = e) = \text{assigns}(x\texttt{:}\ \psi = e) &= \{x\} \\
\text{assigns}(t\ t') &= \text{assigns}(t) \cup \text{assigns}(t') \\
\text{assigns}(\vec{d}) &= \{\vec{f}\} \quad \text{where } d_i = \texttt{def}\ f_i[\vec{\alpha}_i](\vec{x}_i\texttt{:}\ \vec{\psi}_i)\ \texttt{->}\ \psi_i\texttt{:}\ s_i \\
\text{assigns}(\texttt{match}\ e\texttt{:}\ \texttt{case}\ p_1\texttt{:}\ s_1\ \ldots\ \texttt{case}\ p_{n+1}\texttt{:}\ s_{n+1}) &= \textstyle\bigcup_i \left(\text{binds}(p_i) \cup \text{assigns}(s_i)\right) \\
\text{assigns}(\chi_1 \ldots \chi_{n+1}) &= \text{assigns}(\chi_1) \cup \cdots \cup \text{assigns}(\chi_{n+1}) \\
\text{assigns}(\texttt{@dataclass class}\ c[\vec{\alpha}]\ \texttt{:}\ \ \phi) &= \{c\} \\
\text{assigns}(\texttt{@dataclass class}\ c[\vec{\alpha}](q[\vec{\psi}])\ \texttt{:}\ \ \phi) &= \{c\} \\
\text{assigns}(\texttt{type}\ x[\vec{\alpha}] = \psi) &= \{x\}
\end{aligned}$$

binds(p), binds($\vec{p}$), binds(s) — *set of variables*

Variables introduced by pattern p, by the patterns of a sequence $\vec{p}$, by the generators of qualifiers $\vec{g}$, or by the patterns within statement s, not descending into nested functions. On the cases not shown it recurses structurally, taking the union over the immediate subpatterns, and on a statement is the union over the branches of a conditional and empty on the statements not shown.

$$\begin{aligned}
\text{binds}(x) &= \{x\} \\
\text{binds}(p\ \texttt{as}\ x) &= \text{binds}(p) \cup \{x\} \\
\text{binds}(\varepsilon) &= \varnothing \\
\text{binds}(\texttt{if}\ e \cdot \vec{g}) &= \text{binds}(\vec{g}) \\
\text{binds}(\texttt{for}\ x\ \texttt{in}\ e \cdot \vec{g}) &= \{x\} \cup \text{binds}(\vec{g}) \\
\text{binds}(\texttt{match}\ e\texttt{:}\ \texttt{case}\ p_1\texttt{:}\ s_1\ \ldots\ \texttt{case}\ p_{n+1}\texttt{:}\ s_{n+1}) &= \textstyle\bigcup_i \left(\text{binds}(p_i) \cup \text{binds}(s_i)\right) \\
\text{binds}(s\ s') &= \text{binds}(s) \cup \text{binds}(s')
\end{aligned}$$

$\text{declares}(t), \text{declares}(\vec{\iota}), \text{declares}(\vec{\iota}\ t)$ *sequence of variables*

Names declared anywhere in statement t, imports $\vec{\iota}$ or module body $\vec{\iota}\ t$, in textual order and with duplicates, not descending into nested functions. Plain imports with the same root, such as `a.b` and `a.c`, declare the root once. It is empty on the statements not shown and on the empty imports, and is the concatenation over the branches of a conditional and the cases of a match.

$$
\begin{aligned}
\text{declares}(x\colon\ \psi) = \text{declares}(x\colon\ \psi = e) &= x \\
\text{declares}(t\ t') &= \text{declares}(t) \mathbin{+\!\!+} \text{declares}(t') \\
\text{declares}(\vec{d}) &= \vec{f} \\
&\quad \text{where } d_i = \texttt{def}\ f_i[\vec{\alpha}_i](\vec{x}_i\colon \vec{\psi}_i)\ \texttt{->}\ \psi_i\colon\ s_i \\
\text{declares}(\chi_1 \dots \chi_{n+1}) &= \text{declares}(\chi_1) \mathbin{+\!\!+} \cdots \mathbin{+\!\!+} \text{declares}(\chi_{n+1}) \\
\text{declares}(\texttt{@dataclass class}\ c[\vec{\alpha}]\ :\ \phi) &= c \\
\text{declares}(\texttt{@dataclass class}\ c[\vec{\alpha}](q[\vec{\psi}])\ :\ \phi) &= c \\
\text{declares}(\texttt{type}\ x[\vec{\alpha}] = \psi) &= x \\
\\
\text{declares}(\texttt{import}\ (x \cdot \vec{x}) \cdot \vec{\iota}) &= \text{declares}(\vec{\iota}) \\
&\quad \text{if } \texttt{import}\ (x \cdot \vec{x}\,') \in \vec{\iota} \text{ for some } \vec{x}\,' \\
\text{declares}(\texttt{import}\ (x \cdot \vec{x}) \cdot \vec{\iota}) &= x \cdot \text{declares}(\vec{\iota}) \quad \text{(otherwise)} \\
\text{declares}(\texttt{from}\ q\ \texttt{import}\ x_1, \dots, x_{n+1} \cdot \vec{\iota}) &= x_1 \cdot \cdots \cdot x_{n+1} \cdot \text{declares}(\vec{\iota}) \\
\\
\text{declares}(\vec{\iota}\ t) &= \text{declares}(\vec{\iota}) \mathbin{+\!\!+} \text{declares}(t)
\end{aligned}
$$

$\text{type-entry}_\Gamma(q, \vec{\chi}, i), \text{type-entry}_\Gamma(q, \vec{\chi})$ *context entry, or sequence of context entries*

Context entry associated with the name declared by χ_i, a definition of mutual type region $\vec{\chi}$ of module q under context Γ: the class if χ_i is a class definition, and if χ_i is a `type` statement an alias carrying $\vec{\chi}$ and Γ, under which the body of χ_i is resolved at each use (ty-alias). For an entire type region $\vec{\chi}$, the sequence of these entries.

$$
\begin{aligned}
\text{type-entry}_\Gamma(q, \vec{\chi}, i) &= \mathsf{Type}(q.c) && \text{if } \chi_i = \texttt{@dataclass class}\ c[\vec{\alpha}]\ :\ \phi \\
\text{type-entry}_\Gamma(q, \vec{\chi}, i) &= \mathsf{Type}(q.c) && \text{if } \chi_i = \texttt{@dataclass class}\ c[\vec{\alpha}](q'[\vec{\psi}])\ :\ \phi \\
\text{type-entry}_\Gamma(q, \vec{\chi}, i) &= \mathsf{TypeAlias}_\Gamma(q, \vec{\chi}, i) && \text{if } \chi_i = \texttt{type}\ x[\vec{\alpha}] = \psi
\end{aligned}
$$

$$
\text{type-entry}_\Gamma(q, \vec{\chi}) = \text{type-entry}_\Gamma(q, \vec{\chi}, 1) \cdot \ldots \cdot \text{type-entry}_\Gamma(q, \vec{\chi}, n) \qquad \text{where } n = |\vec{\chi}|
$$

$\text{type-context}_\Gamma(q, \vec{\chi}, i)$ *context*

Context under which the type expressions of χ_i, a definition of mutual type region $\vec{\chi}$ of module q, are resolved, given context Γ in which the type region appears: Γ extended with the bindings of every definition of $\vec{\chi}$ if χ_i is a `type` statement, or of the definitions before χ_i if it is a class definition, and then with the type parameters of χ_i.

$$
\begin{aligned}
\text{type-context}_\Gamma(q, \vec{\chi}, i) &= \Gamma \otimes \{\text{declares}(\vec{\chi})\colon \text{type-entry}_\Gamma(q, \vec{\chi})\} \otimes \{\vec{\alpha}\colon \mathsf{TypeVar}\} \\
&\quad \text{if } \chi_i = \texttt{type}\ x[\vec{\alpha}] = \psi \\
\text{type-context}_\Gamma(q, \vec{\chi}, i) &= \Gamma \otimes \Delta_1 \otimes \cdots \otimes \Delta_{i-1} \otimes \{\vec{\alpha}\colon \mathsf{TypeVar}\} \\
&\quad \text{if } \chi_i = \texttt{@dataclass class}\ c[\vec{\alpha}]\ :\ \phi \\
\text{type-context}_\Gamma(q, \vec{\chi}, i) &= \Gamma \otimes \Delta_1 \otimes \cdots \otimes \Delta_{i-1} \otimes \{\vec{\alpha}\colon \mathsf{TypeVar}\} \\
&\quad \text{if } \chi_i = \texttt{@dataclass class}\ c[\vec{\alpha}](q'[\vec{\psi}])\ :\ \phi \\
&\quad \text{where } \Delta_j = \{\text{declares}(\chi_j)\colon \text{type-entry}_\Gamma(q, \vec{\chi}, j)\} \text{ for each } j
\end{aligned}
$$

$\text{class-table}_\Gamma(q, \vec{\chi}, i), \text{class-table}_\Gamma(q, \vec{\chi})$ *class table*

Class table contributed by χ_i, a definition of mutual type region $\vec{\chi}$ of module q under context Γ, which maps the defined class to χ_i under Γ if χ_i is a class definition and is empty if χ_i is a `type` statement. For an entire type region $\vec{\chi}$, the class table defining every class declared in $\vec{\chi}$.

$$\begin{aligned}
\text{class-table}_\Gamma(q, \vec{\chi}, i) &= \{q.c : \mathsf{ClassDef}_\Gamma(q, \vec{\chi}, i)\} && \text{if } \chi_i = \texttt{@dataclass class } c[\vec{\alpha}] : \ \phi \\
\text{class-table}_\Gamma(q, \vec{\chi}, i) &= \{q.c : \mathsf{ClassDef}_\Gamma(q, \vec{\chi}, i)\} && \text{if } \chi_i = \texttt{@dataclass class } c[\vec{\alpha}](q'[\vec{\psi}]) : \ \phi \\
\text{class-table}_\Gamma(q, \vec{\chi}, i) &= \emptyset && \text{(otherwise)}
\end{aligned}$$

$$\text{class-table}_\Gamma(q, \vec{\chi}) = \text{class-table}_\Gamma(q, \vec{\chi}, 1) \otimes \cdots \otimes \text{class-table}_\Gamma(q, \vec{\chi}, n) \qquad \text{where } n = |\vec{\chi}|$$

$\text{scope}(\vec{y}, s)$ *context*

Context for the body s of a function with parameters $\vec{y}$, or of a module with imported names $\vec{y}$, in which each variable declared or assigned in s is unbound and the names in $\vec{y}$ keep their bindings. Undefined if the names $\vec{y}$ are not distinct, if s declares a name twice or declares a name in $\vec{y}$, or if a pattern in s binds a name declared in s or in $\vec{y}$. The context serves as an environment as well, since Unbound is a value as well as a context entry.

$$\begin{aligned}
\text{scope}(\vec{y}, s) = \{x : \mathsf{Unbound} \mid x \in \text{assigns}(s) \setminus \{\vec{y}\}\} \\
\text{if } \vec{y} \text{ distinct, declares}(s) \text{ distinct,} \\
\text{and } \{\vec{y}\}, \{\text{declares}(s)\}, \text{binds}(s) \text{ pairwise disjoint}
\end{aligned}$$

$\text{captures}(e)$ *set of variables*

Variables of the enclosing scope captured by lambdas within expression e; the generator variables of a comprehension are local to it. On the cases not shown it recurses structurally, taking the union over the immediate subexpressions.

$$\begin{aligned}
\text{captures}(\texttt{lambda } \vec{x} : \ e) &= \text{fv}(e) \setminus \{\vec{x}\} \\
\text{captures}([\ e\ \vec{g}\ ]) &= \text{captures}(\vec{g}) \cup (\text{captures}(e) \setminus \text{binds}(\vec{g})) \\
\text{captures}(\{e : \ e'\ \vec{g}\}) &= \text{captures}(\vec{g}) \cup (\text{captures}((e, e')) \setminus \text{binds}(\vec{g}))
\end{aligned}$$

$\text{captures}(\vec{g})$ *set of variables*

Variables of the enclosing scope captured by lambdas within the qualifiers $\vec{g}$ of a comprehension.

$$\begin{aligned}
\text{captures}(\varepsilon) &= \emptyset \\
\text{captures}(\texttt{if } e \cdot \vec{g}) &= \text{captures}(e) \cup \text{captures}(\vec{g}) \\
\text{captures}(\texttt{for } x \texttt{ in } e \cdot \vec{g}) &= \text{captures}(e) \cup (\text{captures}(\vec{g}) \setminus \{x\})
\end{aligned}$$

$\text{fv}(e), \text{fv}(\vec{e}), \text{fv}(\tau)$ *set of variables*

Free variables of expression e, defined in the usual way, and of a sequence of expressions $\vec{e}$, as the union over the expressions; type variables of type τ.

tyvars(Γ) *set of variables*

Type variables in scope in context Γ: the type parameters bound by Γ, and the free type variables of the other entries.

$$\mathrm{tyvars}(\Gamma) = \bigcup\{\mathrm{tyvars}(x : \theta) \mid x : \theta \in \Gamma\}$$

$$\begin{aligned}
\mathrm{tyvars}(x : \mathsf{TypeVar}) &= \{x\} \\
\mathrm{tyvars}(x : \tau) = \mathrm{tyvars}(x : \mathsf{DU}(\tau)) = \mathrm{tyvars}(x : \mathsf{PU}(\tau)) &= \mathrm{fv}(\tau) \\
\mathrm{tyvars}(x : \forall\vec{\alpha}.\ \tau) = \mathrm{tyvars}(x : \mathsf{PU}(\forall\vec{\alpha}.\ \tau)) &= \mathrm{fv}(\tau) \setminus \{\vec{\alpha}\} \\
\mathrm{tyvars}(x : \theta) &= \emptyset \qquad \text{(otherwise)}
\end{aligned}$$

A.2 Functions on types and shapes

The functions on types and shapes introduced in §3.1, each defined by its cases.

base-type(ℓ), base-type(τ) *type*

Base type of literal ℓ, and of type τ that type with a literal type replaced by the base type of its literal.

$$\begin{aligned}
\text{base-type}(n) &= \texttt{int} && \text{if } n \text{ an integer literal} \\
\text{base-type}(n) &= \texttt{float} && \text{if } n \text{ a float literal} \\
\text{base-type}(w) &= \texttt{str} \\
\text{base-type}(\texttt{True}) = \text{base-type}(\texttt{False}) &= \texttt{bool} \\
\text{base-type}(\texttt{None}) &= \texttt{None} \\
\text{base-type}(\texttt{Literal}[\ell]) &= \text{base-type}(\ell) \\
\text{base-type}(\tau) &= \tau && \text{(otherwise)}
\end{aligned}$$

$\sigma \vee_\Sigma \tau$, $\bigvee_\Sigma(\vec{\tau})$ *type*

Larger of two comparable types, and their union otherwise. A sequence joins its first type with the join of the rest, with `Never` as the unit.

$$\begin{aligned}
\sigma \vee_\Sigma \tau &= \tau && \text{if } \Sigma \vdash \sigma <: \tau \\
\sigma \vee_\Sigma \tau &= \sigma && \text{if } \Sigma \vdash \tau <: \sigma \text{ and } \Sigma \nvdash \sigma \equiv \tau \\
\sigma \vee_\Sigma \tau &= \sigma \mid \tau && \text{otherwise}
\end{aligned}$$

$$\begin{aligned}
\textstyle\bigvee_\Sigma(\varepsilon) &= \texttt{Never} \\
\textstyle\bigvee_\Sigma(\sigma \cdot \vec{\tau}) &= \sigma \vee_\Sigma \textstyle\bigvee_\Sigma(\vec{\tau})
\end{aligned}$$

$\sigma \wedge_\Sigma \tau$ *type*

Cases after the first two apply only when σ and τ are incomparable.

$$
\begin{aligned}
\sigma \wedge_\Sigma \tau &= \sigma && \text{if } \Sigma \vdash \sigma <: \tau \\
\sigma \wedge_\Sigma \tau &= \tau && \text{if } \Sigma \vdash \tau <: \sigma \text{ and } \Sigma \nvdash \sigma \equiv \tau \\
\sigma \mid \sigma' \wedge_\Sigma \tau &= \sigma \wedge_\Sigma \tau \vee_\Sigma \sigma' \wedge_\Sigma \tau \\
\sigma \wedge_\Sigma \tau \mid \tau' &= \sigma \wedge_\Sigma \tau \vee_\Sigma \sigma \wedge_\Sigma \tau' && \text{if } \sigma \text{ not union} \\
\texttt{Callable}[[\vec{\sigma}], \sigma] \wedge_\Sigma \texttt{Callable}[[\vec{\tau}], \tau] &= \texttt{Callable}[[\sigma_1 \vee_\Sigma \tau_1, \dots, \sigma_n \vee_\Sigma \tau_n], \sigma \wedge_\Sigma \tau] \\
\texttt{tuple}[\vec{\sigma}] \wedge_\Sigma \texttt{tuple}[\vec{\tau}] &= \texttt{tuple}[\sigma_1 \wedge_\Sigma \tau_1, \dots, \sigma_n \wedge_\Sigma \tau_n] \\
\sigma \wedge_\Sigma \tau &= \texttt{Never} && \text{otherwise}
\end{aligned}
$$

$\text{match-args}(\tau, \sigma), \text{match-args}(\vec{\tau}, \vec{\sigma})$ *context binding type variables at types*

Type arguments for the type variables of τ, found by one-way matching of τ against σ, or of $\vec{\tau}$ against $\vec{\sigma}$. A union in τ does not bind any type variable, since the disjunct matched by the type at the same position in σ is not determined. The matching is undefined for a type variable or union in σ at the position of a type in τ other than a type variable or union, since the two types may or may not be equivalent at some instantiation of the type variables of τ.

$$
\begin{aligned}
\text{match-args}(\alpha, \sigma) &= \{\alpha : \sigma\} \\
\text{match-args}(\tau \mid \tau', \sigma) &= \emptyset \\
\text{match-args}(\texttt{list}[\tau], \texttt{list}[\sigma]) &= \text{match-args}(\tau, \sigma) \\
\text{match-args}(\texttt{dict}[\texttt{str}, \tau], \texttt{dict}[\texttt{str}, \sigma]) &= \text{match-args}(\tau, \sigma) \\
\text{match-args}(\texttt{tuple}[\vec{\tau}], \texttt{tuple}[\vec{\sigma}]) &= \text{match-args}(\vec{\tau}, \vec{\sigma}) \\
\text{match-args}(\texttt{Callable}[[\vec{\tau}], \tau], \texttt{Callable}[[\vec{\sigma}], \sigma]) &= \text{match-args}(\tau \cdot \vec{\tau}, \sigma \cdot \vec{\sigma}) \\
\text{match-args}(C[\vec{\tau}], C[\vec{\sigma}]) &= \text{match-args}(\vec{\tau}, \vec{\sigma}) \\
\text{match-args}(\tau, \tau) &= \emptyset \\
\\
\text{match-args}(\varepsilon, \varepsilon) &= \emptyset \\
\text{match-args}(\tau \cdot \vec{\tau}, \sigma \cdot \vec{\sigma}) &= \text{match-args}(\tau, \sigma) \otimes \text{match-args}(\vec{\tau}, \vec{\sigma})
\end{aligned}
$$

$\text{type-args}_\Sigma(\vec{\alpha}, \sigma, \tau), \text{type-args}_\Sigma(\vec{\alpha}, \vec{\sigma}, \vec{\tau})$ *context binding type variables at types*

Arguments for the type variables $\vec{\alpha}$ of σ, found by one-way unification of σ with τ in which $\vec{\alpha}$ are the unknowns, given as a context binding type variables at types. A variable of $\vec{\alpha}$ takes the base type of τ; a union τ contributes by each disjunct; lists, dictionaries, tuples and callables are taken componentwise; an instantiated class is taken through the instance of the class at τ (Figure 3.4–1); and a union σ contributes nothing when τ is below one of its disjuncts without a variable of $\vec{\alpha}$, as `None` is below `T | None`, and otherwise contributes by each disjunct. In every other case, in particular for a type without type variables, the context is empty. Positions are covariant (tuple components and `Callable` results), contravariant (`Callable` parameters) or invariant (list, dictionary and class arguments), following the subtyping rules (Figure 3.1–5). The type argument for a type variable must accordingly be a supertype, a subtype or an equivalent of the type found, which the argument check of the call

rule enforces once the type arguments are chosen. Contexts combine by pointwise join, written $\vee_\Sigma$ as for types.

$$
\begin{aligned}
\text{type-args}_\Sigma(\vec{\alpha}, \alpha_i, \tau) &= \{\alpha_i : \text{base-type}(\tau)\} \\
\text{type-args}_\Sigma(\vec{\alpha}, \sigma, \tau \mid \tau') &= \text{type-args}_\Sigma(\vec{\alpha}, \sigma, \tau) \vee_\Sigma \text{type-args}_\Sigma(\vec{\alpha}, \sigma, \tau') \\
\text{type-args}_\Sigma(\vec{\alpha}, \texttt{list}[\sigma], \texttt{list}[\tau]) &= \text{type-args}_\Sigma(\vec{\alpha}, \sigma, \tau) \\
\text{type-args}_\Sigma(\vec{\alpha}, \texttt{dict}[\texttt{str}, \sigma], \texttt{dict}[\texttt{str}, \tau]) &= \text{type-args}_\Sigma(\vec{\alpha}, \sigma, \tau) \\
\text{type-args}_\Sigma(\vec{\alpha}, \texttt{tuple}[\vec{\sigma}], \texttt{tuple}[\vec{\tau}]) &= \text{type-args}_\Sigma(\vec{\alpha}, \vec{\sigma}, \vec{\tau}) \\
\text{type-args}_\Sigma(\vec{\alpha}, \texttt{Callable}[[\vec{\sigma}], \sigma], \texttt{Callable}[[\vec{\tau}], \tau]) &= \text{type-args}_\Sigma(\vec{\alpha}, \sigma \cdot \vec{\sigma}, \tau \cdot \vec{\tau}) \\
\text{type-args}_\Sigma(\vec{\alpha}, C[\vec{\sigma}], \tau) &= \text{type-args}_\Sigma(\vec{\alpha}, \vec{\sigma}, \vec{\tau}') \\
&\qquad \text{if } \Sigma \vdash C[\vec{\tau}'] \asymp \tau \\
\text{type-args}_\Sigma(\vec{\alpha}, \sigma \mid \sigma', \tau) &= \emptyset \\
&\qquad \text{if } \Sigma \vdash \tau <: \sigma \text{ and } \text{fv}(\sigma) \cap \{\vec{\alpha}\} = \emptyset \\
\text{type-args}_\Sigma(\vec{\alpha}, \sigma \mid \sigma', \tau) &= \emptyset \\
&\qquad \text{if } \Sigma \vdash \tau <: \sigma' \text{ and } \text{fv}(\sigma') \cap \{\vec{\alpha}\} = \emptyset \\
\text{type-args}_\Sigma(\vec{\alpha}, \sigma \mid \sigma', \tau) &= \text{type-args}_\Sigma(\vec{\alpha}, \sigma, \tau) \vee_\Sigma \text{type-args}_\Sigma(\vec{\alpha}, \sigma', \tau) \quad \text{(otherwise)} \\
\text{type-args}_\Sigma(\vec{\alpha}, \sigma, \tau) &= \emptyset \quad \text{(otherwise)}
\end{aligned}
$$

$$
\begin{aligned}
\text{type-args}_\Sigma(\vec{\alpha}, \varepsilon, \varepsilon) &= \emptyset \\
\text{type-args}_\Sigma(\vec{\alpha}, \sigma \cdot \vec{\sigma}, \tau \cdot \vec{\tau}) &= \text{type-args}_\Sigma(\vec{\alpha}, \sigma, \tau) \vee_\Sigma \text{type-args}_\Sigma(\vec{\alpha}, \vec{\sigma}, \vec{\tau})
\end{aligned}
$$

$$
\begin{aligned}
(\Gamma \vee_\Sigma \Gamma')(\alpha) &= \Gamma(\alpha) \vee_\Sigma \Gamma'(\alpha) && \text{if } \alpha \in \text{dom}(\Gamma) \cap \text{dom}(\Gamma') \\
(\Gamma \vee_\Sigma \Gamma')(\alpha) &= \Gamma(\alpha) && \text{if } \alpha \in \text{dom}(\Gamma) \setminus \text{dom}(\Gamma') \\
(\Gamma \vee_\Sigma \Gamma')(\alpha) &= \Gamma'(\alpha) && \text{if } \alpha \in \text{dom}(\Gamma') \setminus \text{dom}(\Gamma)
\end{aligned}
$$

$\text{elem-type}_\Sigma(\tau)$ — *type*

Type of the elements drawn by a generator from a list, string, dictionary, tuple or range type or a union of these, mirroring iter (§A.4). The element type of a tuple is the join of the base types of its components. Undefined at any other type.

$$
\begin{aligned}
\text{elem-type}_\Sigma(\texttt{list}[\tau]) &= \tau \\
\text{elem-type}_\Sigma(\texttt{str}) &= \texttt{str} \\
\text{elem-type}_\Sigma(\texttt{Literal}[w]) &= \texttt{str} \\
\text{elem-type}_\Sigma(\texttt{dict}[\texttt{str}, \tau]) &= \texttt{str} \\
\text{elem-type}_\Sigma(\texttt{builtins.range}[]) &= \texttt{int} \\
\text{elem-type}_\Sigma(\texttt{tuple}[\vec{\tau}]) &= \bigvee\nolimits_\Sigma(\text{base-type}(\tau_1), \ldots, \text{base-type}(\tau_n)) \\
\text{elem-type}_\Sigma(\sigma \mid \tau) &= \text{elem-type}_\Sigma(\sigma) \vee_\Sigma \text{elem-type}_\Sigma(\tau)
\end{aligned}
$$

$\text{shapes}_\Sigma(\tau, H), \text{shapes}_\Sigma(\vec{\tau})$ — *ordered set of shapes or of shape sequences*

Shapes of type τ that remain once the heads in H are excluded. A union has no head to exclude, so the disjuncts of the union are taken separately, each with the heads of H typed at that disjunct. A dictionary is refined by its keys rather than by a head, so a dictionary type gives the dictionary form with no key bound. Every other type gives the rest form $\tau \setminus H$, except that no shape remains when H exhausts τ: a literal type whose literal is in H, `bool` when both `True` and `False` are in H, `None` when `None` is in H, an instantiated class with an ancestor in H,

and `Never`. A sequence of types yields the sequences of shapes of its components.

$$
\begin{aligned}
\text{shapes}_\Sigma(\sigma \mid \tau, H) &= \text{shapes}_\Sigma(\sigma, H_\sigma) \cup \text{shapes}_\Sigma(\tau, H_\tau) \\
&\qquad \text{where } H_\sigma = \{h \in H \mid \Sigma \vdash h : \sigma\} \text{ and } H_\tau = \{h \in H \mid \Sigma \vdash h : \tau\} \\
\text{shapes}_\Sigma(\texttt{Literal}[\ell], H) &= \emptyset && \text{if } \ell \in H \\
\text{shapes}_\Sigma(\texttt{bool}, H) &= \emptyset && \text{if } \{\texttt{True}, \texttt{False}\} \subseteq H \\
\text{shapes}_\Sigma(\texttt{None}, H) &= \emptyset && \text{if } \texttt{None} \in H \\
\text{shapes}_\Sigma(C[\vec{\tau}], H) &= \emptyset && \text{if } H \cap \text{ancestors}_\Sigma(C) \neq \emptyset \\
\text{shapes}_\Sigma(\texttt{Never}, H) &= \emptyset \\
\text{shapes}_\Sigma(\texttt{dict}[\texttt{str}, \tau], H) &= \{\emptyset \setminus_\tau \emptyset\} \\
\text{shapes}_\Sigma(\tau, H) &= \{\tau \setminus H\} && \text{otherwise} \\
\text{shapes}_\Sigma(\vec{\tau}) &= \text{shapes}_\Sigma(\tau_1, \emptyset) \times \cdots \times \text{shapes}_\Sigma(\tau_n, \emptyset)
\end{aligned}
$$

shape-type(k) *type*

Least type of shape k, with $\Sigma \vdash k : \text{shape-type}(k)$ (Figure 3.4–3c): the instantiated class of a constructor, componentwise for a tuple, and the type recorded by the form otherwise.

$$
\begin{aligned}
\text{shape-type}(\tau \setminus H) &= \tau \\
\text{shape-type}(C[\vec{\tau}](\vec{k}) \setminus H) &= C[\vec{\tau}] \\
\text{shape-type}((\vec{k})) &= \texttt{tuple}[\text{shape-type}(k_1), \ldots, \text{shape-type}(k_n)] \\
\text{shape-type}([\vec{k}]_\tau) &= \texttt{list}[\tau] \\
\text{shape-type}(\beta \setminus_\tau H) &= \texttt{dict}[\texttt{str}, \tau]
\end{aligned}
$$

matched-as$_\Sigma(\Gamma, \tau, p)$ *type*

Type to which τ is narrowed if pattern p matches: an over-approximation of the matched values, as a type. Variable and wildcard patterns leave τ unchanged; literal patterns leave a type variable unchanged, since a value of the type variable may equal the literal even though the meet of a literal type with a type variable is `Never`; float literal patterns narrow to `float`, and other literal patterns to their literal types; tuple patterns narrow each component; list and dictionary patterns narrow to the list or dictionary type; constructor patterns narrow to the instance of their class at τ, or to τ itself when τ is below the instance; unions are narrowed disjunct by disjunct. Any other pattern does not match any value of τ, and narrows to `Never`. The function emulates the narrowing

of a match scrutinee observed in mypy within a case, and gives the types of patterns named by `as` (check-as).

$$
\begin{aligned}
\text{matched-as}_\Sigma(\Gamma, \sigma \mid \tau, p) &= \text{matched-as}_\Sigma(\Gamma, \sigma, p) \vee_\Sigma \text{matched-as}_\Sigma(\Gamma, \tau, p) \\
\text{matched-as}_\Sigma(\Gamma, \tau, x) &= \tau \\
\text{matched-as}_\Sigma(\Gamma, \tau, _) &= \tau \\
\text{matched-as}_\Sigma(\Gamma, \tau, p \texttt{ as } x) &= \text{matched-as}_\Sigma(\Gamma, \tau, p) \\
\text{matched-as}_\Sigma(\Gamma, \alpha, \ell) &= \alpha \\
\text{matched-as}_\Sigma(\Gamma, \tau, n) &= \texttt{float} \wedge_\Sigma \tau \\
&\qquad \text{if } n \text{ a float literal} \\
\text{matched-as}_\Sigma(\Gamma, \tau, \ell) &= \texttt{Literal}[\ell] \wedge_\Sigma \tau \\
\text{matched-as}_\Sigma(\Gamma, \texttt{tuple}[\tau_1, \ldots, \tau_n], (p_1, \ldots, p_n)) &= \texttt{tuple}[\sigma_1, \ldots, \sigma_n] \\
&\qquad \text{where } \sigma_i = \text{matched-as}_\Sigma(\Gamma, \tau_i, p_i) \quad (\forall i) \\
\text{matched-as}_\Sigma(\Gamma, \texttt{list}[\tau], [\vec{p}]) &= \texttt{list}[\tau] \\
\text{matched-as}_\Sigma(\Gamma, \texttt{dict}[\texttt{str}, \tau], \{\vec{w} \colon \vec{p}\}) &= \texttt{dict}[\texttt{str}, \tau] \\
\text{matched-as}_\Sigma(\Gamma, \tau, q(a)) &= \tau \wedge_\Sigma C[\vec{\sigma}] \\
&\qquad \text{if } \Gamma \vdash q \,:\, \mathsf{Type}(C) \text{ and } \Sigma \vdash C[\vec{\sigma}] \asymp \tau \\
\text{matched-as}_\Sigma(\Gamma, \tau, p) &= \texttt{Never} \quad \text{(otherwise)}
\end{aligned}
$$

$\text{remaining}_\Sigma(\Gamma, \tau, p)$ *type*

Type remaining from τ if pattern p fails to match: an over-approximation of the unmatched values, as a type rather than a set of shapes. Nothing remains after a variable or wildcard pattern, after a literal pattern at its own literal type or at `None`, after the empty dictionary pattern at a dictionary type, after a tuple pattern each of whose sub-patterns leaves nothing, or after a constructor pattern at a type below the instance of its class each of whose sub-patterns leaves nothing at the field types. The other Boolean literal remains after a literal pattern at `bool`; tuple patterns with exactly one sub-pattern that leaves something narrow that component; unions are narrowed disjunct by disjunct; and any other pattern leaves τ unchanged. The function emulates the narrowing of a match scrutinee observed in mypy, so that a match which is exhaustive by the remaining type is also exhaustive under mypy. The remaining type is coarser than the residual (§3.4), since constructor patterns

with a refutable sub-pattern, and tuple patterns with two refutable components, leave the type untouched.

$$
\begin{aligned}
\text{remaining}_\Sigma(\Gamma, \sigma \mid \tau, p) &= \text{remaining}_\Sigma(\Gamma, \sigma, p) \vee_\Sigma \text{remaining}_\Sigma(\Gamma, \tau, p) \\
\text{remaining}_\Sigma(\Gamma, \tau, x) &= \texttt{Never} \\
\text{remaining}_\Sigma(\Gamma, \tau, _) &= \texttt{Never} \\
\text{remaining}_\Sigma(\Gamma, \tau, p \texttt{ as } x) &= \text{remaining}_\Sigma(\Gamma, \tau, p) \\
\text{remaining}_\Sigma(\Gamma, \texttt{Literal}[\ell], \ell) &= \texttt{Never} \\
\text{remaining}_\Sigma(\Gamma, \texttt{bool}, \texttt{True}) &= \texttt{Literal}[\texttt{False}] \\
\text{remaining}_\Sigma(\Gamma, \texttt{bool}, \texttt{False}) &= \texttt{Literal}[\texttt{True}] \\
\text{remaining}_\Sigma(\Gamma, \texttt{None}, \texttt{None}) &= \texttt{Never} \\
\text{remaining}_\Sigma(\Gamma, \texttt{tuple}[\vec{\tau}], (\vec{p})) &= \texttt{Never} \\
&\qquad \text{if } \text{remaining}_\Sigma(\Gamma, \tau_i, p_i) = \texttt{Never} \quad (\forall i) \\
\text{remaining}_\Sigma(\Gamma, \texttt{tuple}[\vec{\tau}], (\vec{p})) &= \texttt{tuple}[\tau_1, \ldots, \tau_{i-1}, \text{remaining}_\Sigma(\Gamma, \tau_i, p_i), \tau_{i+1}, \ldots, \tau_n] \\
&\qquad \text{if } \text{remaining}_\Sigma(\Gamma, \tau_j, p_j) = \texttt{Never} \quad (\forall j \neq i) \\
\text{remaining}_\Sigma(\Gamma, \texttt{dict}[\texttt{str}, \tau], \{\}) &= \texttt{Never} \\
\text{remaining}_\Sigma(\Gamma, \tau, q(a)) &= \texttt{Never} \\
&\qquad \text{if } \Gamma \vdash q : \mathsf{Type}(C), \Sigma \vdash C[\vec{\sigma}] \asymp \tau, \Sigma \vdash \tau <: C[\vec{\sigma}], \\
&\qquad \text{fields}_\Sigma(C)[\vec{\sigma}] = \vec{x} \colon \vec{\sigma}', \text{field-map}_\Sigma(C, a) = \{\vec{x} \colon \vec{p}\} \\
&\qquad \text{and } \text{remaining}_\Sigma(\Gamma, \sigma'_i, p_i) = \texttt{Never} \quad (\forall i) \\
\text{remaining}_\Sigma(\Gamma, \tau, p) &= \tau
\end{aligned}
$$

$\text{resolve-op}_\Sigma(\odot, \sigma_1, \sigma_2), \text{resolve-op}_\Sigma(\ominus, \sigma)$	*type*

Return type of the most specific overload of an operator at the operand types. The overloads of $\odot$ at (σ_1, σ_2) are the resolved overloads $((\tau_1, \tau_2), \tau)$ with $\Sigma \vdash \odot(\sigma_1, \sigma_2) : ((\tau_1, \tau_2), \tau)$ (Figure 3.2–6), ordered by componentwise subtyping in Σ on their bounds (τ_1, τ_2); the overloads of $\ominus$ at σ are defined in the same way from conclusions $\Sigma \vdash \ominus(\sigma) : (\tau, \tau')$. The most specific overload is the least element under this order, which exists as long as some overload applies and neither operand type is a subtype of `Never` (Lemma 3.3); the function is undefined when no least element exists.

$$
\begin{aligned}
\text{resolve-op}_\Sigma(\odot, \sigma_1, \sigma_2) &= \tau && \text{if } ((\tau_1, \tau_2), \tau) \text{ most specific overload of } \odot \text{ at } (\sigma_1, \sigma_2) \\
\text{resolve-op}_\Sigma(\ominus, \sigma) &= \tau' && \text{if } (\tau, \tau') \text{ most specific overload of } \ominus \text{ at } \sigma
\end{aligned}
$$

A.3 Functions on classes and modules

The functions on module names, field lists and classes introduced in §2.4, and the resolution of names in an environment.

$\text{submods}(q)$	*context*

Context that binds each submodule of q to a stub: a context entry $x : \mathsf{Mod}(q.x)$ for each module $q.x$ of the program or of the predefined modules.

$$\text{submods}(q) = \{x : \mathsf{Mod}(q.x) \mid q.x \in \text{dom}(\mathcal{M}) \cup \text{dom}(\mathcal{P})\}$$

unique-to(q', q) *sequence of module names*

Modules which strictly contain q' but do not contain q, shortest first. A from-import of q' in module q checks these as well as q' itself.

$$\text{unique-to}(q', q) = q_1^\dagger \cdot \cdots \cdot q_m^\dagger \qquad \text{where } \{q_1^\dagger, \ldots, q_m^\dagger\} = \{q^\dagger \prec q' \mid q^\dagger \not\preceq q\},\ q_1^\dagger \prec \cdots \prec q_m^\dagger$$

loads($\vec{q}$, $\vec{q}'$), submod-loads($\vec{q}$, q, $\vec{x}$), import-loads($\vec{q}$, q, $\vec{\imath}$), import-loads($\vec{q}$, q, ι) *sequence of module names*

Modules $\vec{q}$ loaded so far, extended in load order by the modules loaded by $\vec{q}'$, by importing names $\vec{x}$ from q, or by imports $\vec{\imath}$ or import ι of q, excluding predefined modules. Defined for well-formed programs, which do not have import cycles.

$$\begin{aligned}
\text{loads}(\vec{q}, \varepsilon) &= \vec{q} \\
\text{loads}(\vec{q}, q \cdot \vec{q}') &= \text{loads}(\vec{q}, \vec{q}') && \text{if } q \in \{\vec{q}\} \cup \text{dom}(\mathcal{P}) \\
\text{loads}(\vec{q}, q \cdot \vec{q}') &= \text{loads}(\text{import-loads}(\vec{q}, q, \vec{\imath}) \cdot q, \vec{q}') && \text{if } q \notin \{\vec{q}\} \text{ and } \mathcal{M}(q) = \vec{\imath}\ t
\end{aligned}$$

$$\begin{aligned}
\text{submod-loads}(\vec{q}, q, \varepsilon) &= \vec{q} \\
\text{submod-loads}(\vec{q}, q, x \cdot \vec{x}) &= \text{submod-loads}(\text{loads}(\vec{q}, q.x), q, \vec{x}) && \text{if } q, q.x \in \text{dom}(\mathcal{M}) \text{ and } x \notin \text{declares}(\mathcal{M}(q)) \\
\text{submod-loads}(\vec{q}, q, x \cdot \vec{x}) &= \text{submod-loads}(\vec{q}, q, \vec{x}) && \text{(otherwise)}
\end{aligned}$$

$$\begin{aligned}
\text{import-loads}(\vec{q}, q, \varepsilon) &= \vec{q} \\
\text{import-loads}(\vec{q}, q, \iota \cdot \vec{\imath}) &= \text{import-loads}(\text{import-loads}(\vec{q}, q, \iota), q, \vec{\imath}) \\
\text{import-loads}(\vec{q}, q, \texttt{import}\ x_1.\cdots.x_n) &= \text{loads}(\vec{q}, x_1 \cdot x_1.x_2 \cdot \cdots \cdot x_1.\cdots.x_n) \\
\text{import-loads}(\vec{q}, q, \texttt{from}\ q'\ \texttt{import}\ \vec{x}) &= \text{submod-loads}(\text{loads}(\vec{q}, \text{unique-to}(q', q) \cdot q'), q', \vec{x})
\end{aligned}$$

import-env$_{\mathcal{L}}$($\vec{\imath}$), import-env$_{\mathcal{L}}$(ι), submodules-env$_{\mathcal{L}}$(q, $\vec{x}$) *environment*

Environment bound by imports $\vec{\imath}$ or import ι, and environment of module q extended along submodule chain $\vec{x}$, given loaded module environments $\mathcal{L}$.

$$\begin{aligned}
\text{import-env}_{\mathcal{L}}(\varepsilon) &= \emptyset \\
\text{import-env}_{\mathcal{L}}(\iota \cdot \vec{\imath}) &= \text{import-env}_{\mathcal{L}}(\iota) \boxplus \text{import-env}_{\mathcal{L}}(\vec{\imath}) \\
\text{import-env}_{\mathcal{L}}(\texttt{import}\ (x \cdot \vec{x})) &= \{x : \mathsf{Mod}(x, \text{submodules-env}_{\mathcal{L}}(x, \vec{x}))\} \\
\text{import-env}_{\mathcal{L}}(\texttt{from}\ q\ \texttt{import}\ x_1, \ldots, x_{n+1}) &= \{x_1 : \text{member}_{\mathcal{L}}(\mathcal{L}(q), x_1), \ldots, x_{n+1} : \text{member}_{\mathcal{L}}(\mathcal{L}(q), x_{n+1})\}
\end{aligned}$$

$$\begin{aligned}
\text{submodules-env}_{\mathcal{L}}(q, \varepsilon) &= \mathcal{L}(q) \\
\text{submodules-env}_{\mathcal{L}}(q, x \cdot \vec{x}) &= \mathcal{L}(q) \boxplus \{x : \mathsf{Mod}(q.x, \text{submodules-env}_{\mathcal{L}}(q.x, \vec{x}))\}
\end{aligned}$$

member$_{\mathcal{L}}$(ρ, x) *value*

Value of x in module environment ρ, as a loaded module value if a stub, given loaded module environments $\mathcal{L}$.

$$\begin{aligned}
\text{member}_{\mathcal{L}}(\rho, x) &= \mathsf{Mod}(q, \mathcal{L}(q)) && \text{if } \rho(x) = \mathsf{Mod}(q) \\
\text{member}_{\mathcal{L}}(\rho, x) &= \rho(x) && \text{(otherwise)}
\end{aligned}$$

own-fields(ϕ), own-fields(χ) — *sequence of bindings*

Fields declared by a field list, or by a class definition, with their types, in order.

$$\begin{aligned}
\text{own-fields}(x_1 \colon \psi_1 \ \ldots \ x_{n+1} \colon \psi_{n+1}) &= x_1 \colon \psi_1, \ldots, x_{n+1} \colon \psi_{n+1} \\
\text{own-fields}(\texttt{pass}) &= \varepsilon \\
\text{own-fields}(\texttt{@dataclass class } c[\vec{\alpha}] \ : \ \phi) &= \text{own-fields}(\phi) \\
\text{own-fields}(\texttt{@dataclass class } c[\vec{\alpha}](q[\vec{\psi}]) \ : \ \phi) &= \text{own-fields}(\phi)
\end{aligned}$$

type-params$_\Sigma(C)$ — *sequence of type parameters*

Type parameters of class C, as declared.

$$\begin{aligned}
&\text{type-params}_\Sigma(C) = \vec{\alpha} && \text{if } \Sigma(C) = \mathsf{ClassDef}_\Gamma(q, \vec{\chi}, i) \text{ and } \chi_i = \texttt{@dataclass class } c[\vec{\alpha}] \ : \ \phi \\
&\text{type-params}_\Sigma(C) = \vec{\alpha} && \text{if } \Sigma(C) = \mathsf{ClassDef}_\Gamma(q, \vec{\chi}, i) \text{ and } \chi_i = \texttt{@dataclass class } c[\vec{\alpha}](q'[\vec{\psi}]) \ : \ \phi
\end{aligned}$$

base$_\Sigma(C)$ — *instantiated class or* $\bot$

Base class of C at the type arguments given in its definition, resolved under the context of the definition, and $\bot$ otherwise.

$$\begin{aligned}
&\text{base}_\Sigma(C) = \bot && \text{if } \chi_i = \texttt{@dataclass class } c[\vec{\alpha}] \ : \ \phi \\
&\text{base}_\Sigma(C) = D[\vec{\sigma}] && \text{if } \chi_i = \texttt{@dataclass class } c[\vec{\alpha}](q'[\vec{\psi}]) \ : \ \phi \\
& && \text{and } \Sigma; \text{type-context}_\Gamma(q, \vec{\chi}, i) \vdash q'[\vec{\psi}] \rhd D[\vec{\sigma}] \\
& && \text{where } \Sigma(C) = \mathsf{ClassDef}_\Gamma(q, \vec{\chi}, i)
\end{aligned}$$

fields$_\Sigma(C)$ — *field scheme*

Fields of class C with their declared types, as a field scheme over the type parameters of C: the fields of its base class, if any, instantiated at the type arguments of the base, followed by the fields declared by C, distinct from the inherited fields by class-extend (Figure 3.3–5). The fields of an instantiated class $C[\vec{\tau}]$ are the instantiation $\text{fields}_\Sigma(C)[\vec{\tau}]$.

$$\begin{aligned}
&\text{fields}_\Sigma(C) = \forall \vec{\alpha}. \, \vec{x} \colon \vec{\tau} && \text{if } \text{base}_\Sigma(C) = \bot \\
&\text{fields}_\Sigma(C) = \forall \vec{\alpha}. \, (\text{fields}_\Sigma(D)[\vec{\sigma}] +\!\!+ \vec{x} \colon \vec{\tau}) && \text{if } \text{base}_\Sigma(C) = D[\vec{\sigma}] \\
& && \text{where } \Sigma(C) = \mathsf{ClassDef}_\Gamma(q, \vec{\chi}, i), \ \vec{\alpha} = \text{type-params}_\Sigma(C), \\
& && \text{own-fields}(\chi_i) = \vec{x} \colon \vec{\psi} \\
& && \text{and } \Sigma; \text{type-context}_\Gamma(q, \vec{\chi}, i) \vdash \vec{\psi} \rhd \vec{\tau}
\end{aligned}$$

ancestors$_\Sigma(C)$ — *set of classes*

Class C together with all its base classes.

$$\begin{aligned}
&\text{ancestors}_\Sigma(C) \ = \{C\} && \text{if } \text{base}_\Sigma(C) = \bot \\
&\text{ancestors}_\Sigma(C) \ = \{C\} \cup \text{ancestors}_\Sigma(D) && \text{if } \text{base}_\Sigma(C) = D[\vec{\sigma}]
\end{aligned}$$

ancestor-args$_\Sigma(C[\vec{\tau}], D)$ *sequence of types*

Type arguments of the instantiation of D above $C[\vec{\tau}]$, which is unique because type arguments are invariant; defined only when $D \in \text{ancestors}_\Sigma(C)$.

$$\begin{aligned}\text{ancestor-args}_\Sigma(C[\vec{\tau}], C) &= \vec{\tau}\\ \text{ancestor-args}_\Sigma(C[\vec{\tau}], D) &= \text{ancestor-args}_\Sigma(E[[\vec{\tau}/\vec{\alpha}]\vec{\sigma}], D)\\ &\qquad \text{if } D \neq C,\ \text{base}_\Sigma(C) = E[\vec{\sigma}] \text{ and } \text{type-params}_\Sigma(C) = \vec{\alpha}\end{aligned}$$

field-map$_\Sigma(C, \vec{u})$ *map from variables*

Argument supplied for each field of C: positional arguments fill the fields in order and keyword arguments match by name. Defined if and only if the arguments cover the fields exactly, so constructor patterns and expressions check and evaluate field by field without regard to the split of the arguments between positional and keyword.

$$\begin{aligned}\text{field-map}_\Sigma(C, (u_1, \ldots, u_n, y_1{=}u_{n+1}, \ldots, y_m{=}u_{n+m})) = \{x_1 : u_1, \ldots, x_n : u_n, y_1 : u_{n+1}, \ldots, y_m : u_{n+m}\}\\ \text{if } \text{fields}_\Sigma(C) = \forall\vec{\alpha}.\, x_1 : \sigma_1 \cdot \ldots \cdot x_n : \sigma_n \cdot \vec{y}\,' : \vec{\sigma}\,'\\ \text{and } \{\vec{y}\} = \{\vec{y}\,'\}\end{aligned}$$

resolve(ρ, q) *value*

Value denoted by name q in ρ, mirroring static judgement $\Gamma \vdash q : \theta$ (Figure 3.1–1): a simple name is looked up in ρ directly, and a qualified name in the environment of the loaded module denoted by its prefix. Undefined if the prefix denotes a module stub or a value other than a module.

$$\begin{aligned}\text{resolve}(\rho, x) &= \rho(x)\\ \text{resolve}(\rho, q.x) &= \rho'(x) \qquad \text{where } \text{resolve}(\rho, q) = \mathsf{Mod}(q', \rho')\end{aligned}$$

name(q) *string*

String literal naming module q, so that name(`__main__`) = `"__main__"`.

A.4 Functions on values

The functions on values used by the evaluation rules. The comparisons `==`, `!=`, `in` and `not in` compare values structurally, by structural equality eq_Σ and membership in_Σ, and agree with Python except at the comparisons left undefined by PurePy (§B.1.3): between a Boolean and a number, between two functions, and between `nan` and `nan` inside a container. A run that reaches one of these comparisons has undefined behaviour, so an implementation may abort or produce the result given by Python (Chapter 1). Equality and membership compare the elements of a sequence in order and stop at the first element that decides the result, so an undefined comparison later in the sequence does not make the whole comparison undefined. Membership in a range, or of a key other than a string in a dictionary, is ruled out by typing and aborts with excluded (§A.4).

type-of(v) *type*

Run-time type of v: `float` for a float literal, the literal type of any other literal, the type of a list, tuple or

dictionary with every element type `object`, and `object` for any other value.

$$\begin{aligned}
\text{type-of}(n) &= \texttt{float} && \text{if } n \text{ a float literal} \\
\text{type-of}(\ell) &= \texttt{Literal}[\ell] && \text{(otherwise)} \\
\\
\text{type-of}([\vec{v}]) &= \texttt{list}[\texttt{object}] \\
\text{type-of}((v_1, \ldots, v_n)) &= \texttt{tuple}[\texttt{object}^n] \\
\text{type-of}(\{\delta\}) &= \texttt{dict}[\texttt{str}, \texttt{object}] \\
\text{type-of}(v) &= \texttt{object} && \text{(otherwise)}
\end{aligned}$$

real(n) *extended real number*

Extended real number denoted by number n other than nan. A condition which applies real to nan does not hold.

$$\begin{aligned}
\text{real}(n) &= n && \text{if } n \text{ an integer} \\
\text{real}(\mathsf{inf}) &= +\infty \\
\text{real}(-\mathsf{inf}) &= -\infty \\
\text{real}(n) &= \text{real number represented by } n \text{ in IEEE 754} && \text{otherwise}
\end{aligned}$$

fl(x) *float*

Real number x rounded to float, provided as shorthand for rounding attribute roundTiesToEven of IEEE 754.

$$\text{fl}(x) = \text{roundTiesToEven}(x)$$

float(x) *result*

Real number x rounded to float by fl, with aborts(OverflowError) in place of infinity, following conversion of integer to float in Python.

$$\begin{aligned}
\text{float}(x) &= \mathsf{aborts}(\mathsf{OverflowError}) && \text{if } \text{fl}(x) \in \{\mathsf{inf}, -\mathsf{inf}\} \\
\text{float}(x) &= \mathsf{val}(\text{fl}(x)) && \text{otherwise}
\end{aligned}$$

trunc(x) *integer*

Integer part of real number x, rounding toward zero.

$$\begin{aligned}
\text{trunc}(x) &= \lfloor x \rfloor && \text{if } x \geq 0 \\
\text{trunc}(x) &= \lceil x \rceil && \text{otherwise}
\end{aligned}$$

floordiv(n, n') *float*

Floor division of float n by nonzero float n', with the algorithm of the Python Language Reference, §6.7, note 2. For n and n' other than nan, let $x = \text{real}(n)$ and $x' = \text{real}(n')$. For finite n and n', let $x_1 = x - (x' \times \text{trunc}(x/x'))$, the remainder of truncated division computed exactly, $n_2 = \text{fl}(\text{real}(\text{fl}(x - x_1))/x')$, and $n_3 = \text{fl}(\text{real}(n_2) - 1)$ if

$x_1 \neq 0$ has the opposite sign to x', and n_2 otherwise.

$\text{floordiv}(n, n') = \mathsf{nan}$	if $n \in \{\mathsf{nan}, \mathsf{inf}, -\mathsf{inf}\}$ or $n' = \mathsf{nan}$
$\text{floordiv}(n, n') = -1.0$	if $n' = \mathsf{inf}$ and $x < 0$
$\text{floordiv}(n, n') = -1.0$	if $n' = -\mathsf{inf}$ and $x > 0$
$\text{floordiv}(n, n') = -0.0$	if $n' = \mathsf{inf}$ and $n = -0.0$
$\text{floordiv}(n, n') = -0.0$	if $n' = -\mathsf{inf}$ and $n = 0.0$
$\text{floordiv}(n, n') = 0.0$	if $n' = \mathsf{inf}$ and ($x > 0$ or $n = 0.0$)
$\text{floordiv}(n, n') = 0.0$	if $n' = -\mathsf{inf}$ and ($x < 0$ or $n = -0.0$)
$\text{floordiv}(n, n') = n_3$	if $n_3 \in \{\mathsf{inf}, -\mathsf{inf}\}$
$\text{floordiv}(n, n') = -0.0$	if $\lceil \text{real}(n_3) - \frac{1}{2} \rceil = 0$ and n, n' of opposite signs
$\text{floordiv}(n, n') = \text{fl}(\lceil \text{real}(n_3) - \frac{1}{2} \rceil)$	otherwise

$\text{mod}(n, n')$ *float*

Remainder of division of float n by nonzero float n', computed exactly and rounded to nearest float. For n and n' other than nan, let $x = \text{real}(n)$ and $x' = \text{real}(n')$. For finite n and n', let $n_1 = \text{fl}(x - (x' \times \lfloor x/x' \rfloor))$.

$\text{mod}(n, n') = \mathsf{nan}$	if $n \in \{\mathsf{nan}, \mathsf{inf}, -\mathsf{inf}\}$ or $n' = \mathsf{nan}$
$\text{mod}(n, n') = n$	if $n' = \mathsf{inf}$ and $x > 0$
$\text{mod}(n, n') = n$	if $n' = -\mathsf{inf}$ and $x < 0$
$\text{mod}(n, n') = n'$	if $n' = \mathsf{inf}$ and $x < 0$
$\text{mod}(n, n') = n'$	if $n' = -\mathsf{inf}$ and $x > 0$
$\text{mod}(n, n') = 0.0$	if $n' = \mathsf{inf}$ and n zero
$\text{mod}(n, n') = -0.0$	if $n' = -\mathsf{inf}$ and n zero
$\text{mod}(n, n') = -0.0$	if $n_1 = 0.0$ and $x' < 0$
$\text{mod}(n, n') = n_1$	otherwise

$\text{eq}_\Sigma(v, v')$ *result*

Structural equality: literals by value, numbers by converting to extended reals, lists and tuples by their elements, dictionaries with the same keys by their values, and objects of the same class by their field values, each through eq-elems_Σ, except ranges, which are compared by their sequences, so that `range(0, -1) == range(0, -2)`. Any other pair of values is unequal, except that equality is undefined between two functions and between nan and nan in a container, which Python compares by identity, and between a Boolean and a number, which Python compares numerically. Compared directly, nan is unequal to itself in both languages (Figure 4.2–1). Dictionaries

are compared in the entry order of the left operand.

$$\begin{aligned}
\mathrm{eq}_\Sigma(n, n') &= \mathsf{val}(\texttt{True}) && \text{if } \mathrm{real}(n) = \mathrm{real}(n') \\
\\
\mathrm{eq}_\Sigma(n, n') &= \mathsf{aborts}(\mathsf{undef}) && \text{if } n = n' = \mathsf{nan} \\
\mathrm{eq}_\Sigma(w, w') &= \mathsf{val}(\texttt{True}) && \text{if } w = w' \\
\mathrm{eq}_\Sigma(w, w') &= \mathsf{val}(\texttt{False}) && \text{if } w \neq w' \\
\mathrm{eq}_\Sigma(\texttt{True}, \texttt{True}) &= \mathrm{eq}_\Sigma(\texttt{False}, \texttt{False}) = \mathsf{val}(\texttt{True}) \\
\mathrm{eq}_\Sigma(\texttt{True}, \texttt{False}) &= \mathrm{eq}_\Sigma(\texttt{False}, \texttt{True}) = \mathsf{val}(\texttt{False}) \\
\mathrm{eq}_\Sigma(\texttt{None}, \texttt{None}) &= \mathsf{val}(\texttt{True}) \\
\mathrm{eq}_\Sigma(\texttt{True}, n) &= \mathrm{eq}_\Sigma(\texttt{False}, n) = \mathsf{aborts}(\mathsf{undef}) \\
\mathrm{eq}_\Sigma(n, \texttt{True}) &= \mathrm{eq}_\Sigma(n, \texttt{False}) = \mathsf{aborts}(\mathsf{undef}) \\
\\
\mathrm{eq}_\Sigma([\vec{v}], [\vec{v}\,']) &= \text{eq-elems}_\Sigma(\vec{v}, \vec{v}\,') \\
\mathrm{eq}_\Sigma((\vec{v}), (\vec{v}\,')) &= \text{eq-elems}_\Sigma(\vec{v}, \vec{v}\,') \\
\\
\mathrm{eq}_\Sigma(\vec{w} : \vec{v}, \delta') &= \text{eq-elems}_\Sigma(\vec{v}, \delta'(\vec{w})) && \text{if } \{\vec{w}\} = \mathrm{dom}(\delta') \\
\mathrm{eq}_\Sigma(\delta, \delta') &= \mathsf{val}(\texttt{False}) && \text{if } \mathrm{dom}(\delta) \neq \mathrm{dom}(\delta') \\
\\
\mathrm{eq}_\Sigma(\mathsf{Obj}(C, \vec{x} : \vec{v}), \mathsf{Obj}(C, \vec{x} : \vec{v}\,')) &= \text{eq-elems}_\Sigma(\mathrm{iter}(\mathsf{Obj}(C, \vec{x} : \vec{v})), \mathrm{iter}(\mathsf{Obj}(C, \vec{x} : \vec{v}\,'))) \\
& \qquad \text{if } C = \texttt{builtins.range} \\
\mathrm{eq}_\Sigma(\mathsf{Obj}(C, \vec{x} : \vec{v}), \mathsf{Obj}(C, \vec{x} : \vec{v}\,')) &= \text{eq-elems}_\Sigma(\vec{v}, \vec{v}\,') \\
& \qquad \text{if } C \neq \texttt{builtins.range} \\
\\
\mathrm{eq}_\Sigma(v, v') &= \mathsf{aborts}(\mathsf{undef}) \\
& \qquad \text{if } v \text{ and } v' \text{ closures or primitives} \\
\mathrm{eq}_\Sigma(v, v') &= \mathsf{val}(\texttt{False}) \quad \text{(otherwise)}
\end{aligned}$$

$\text{eq-elems}_\Sigma(\vec{v}, \vec{v}\,')$	*result*

Equality of two sequences of values: `False` if the lengths differ, and otherwise compared from the first element, stopping at the first pair that is not equal, with that pair's result.

$$\begin{aligned}
\text{eq-elems}_\Sigma(\vec{v}, \vec{v}\,') &= \mathsf{val}(\texttt{False}) && \text{if } |\vec{v}| \neq |\vec{v}\,'| \\
\text{eq-elems}_\Sigma(\varepsilon, \varepsilon) &= \mathsf{val}(\texttt{True}) \\
\text{eq-elems}_\Sigma(v \cdot \vec{v}, v' \cdot \vec{v}\,') &= \text{eq-elems}_\Sigma(\vec{v}, \vec{v}\,') && \text{if } \mathrm{eq}_\Sigma(v, v') = \mathsf{val}(\texttt{True}) \\
\text{eq-elems}_\Sigma(v \cdot \vec{v}, v' \cdot \vec{v}\,') &= \mathrm{eq}_\Sigma(v, v') \quad \text{(otherwise)}
\end{aligned}$$

$\mathrm{in}_\Sigma(v, v')$	*result*

Whether v is an element of v': of a list or tuple by structural equality, of a dictionary as a key, and of a string as a substring. Aborts with excluded when v' is a range, since no overload of `in` accepts a range, although Python allows it; and for a dictionary and a key other than a string (Chapter 4). Aborts with TypeError for any other

pair of values.

$$\begin{aligned}
\text{in}_\Sigma(v, [\vec{v}]) &= \text{in-elems}_\Sigma(v, \vec{v}) \\
\text{in}_\Sigma(v, (\vec{v})) &= \text{in-elems}_\Sigma(v, \vec{v}) \\
\\
\text{in}_\Sigma(w, \delta) &= \mathsf{val}(\texttt{True}) && \text{if } w \in \text{dom}(\delta) \\
\text{in}_\Sigma(w, \delta) &= \mathsf{val}(\texttt{False}) && \text{if } w \notin \text{dom}(\delta) \\
\text{in}_\Sigma(v, \delta) &= \mathsf{aborts}(\text{excluded}) && \text{if } v \text{ not a string} \\
\\
\text{in}_\Sigma(w, w') &= \mathsf{val}(\texttt{True}) && \text{if } w' = w_1 \mathbin{+\!\!+} w \mathbin{+\!\!+} w_2 \text{ for some } w_1, w_2 \\
\text{in}_\Sigma(w, w') &= \mathsf{val}(\texttt{False}) && \text{if there are no such } w_1, w_2 \\
\text{in}_\Sigma(v, v') &= \mathsf{aborts}(\text{excluded}) && \text{if } v' \text{ a range} \\
\text{in}_\Sigma(v, v') &= \mathsf{aborts}(\text{TypeError}) && \text{otherwise}
\end{aligned}$$

$\text{in-elems}_\Sigma(v, \vec{v})$	*result*

Whether a sequence of values has an element structurally equal to v, taking the elements in order and stopping at the first element that is not unequal to v, with that element's result.

$$\begin{aligned}
\text{in-elems}_\Sigma(v, \varepsilon) &= \mathsf{val}(\texttt{False}) \\
\text{in-elems}_\Sigma(v, v' \cdot \vec{v}) &= \text{in-elems}_\Sigma(v, \vec{v}) && \text{if } \text{eq}_\Sigma(v, v') = \mathsf{val}(\texttt{False}) \\
\text{in-elems}_\Sigma(v, v' \cdot \vec{v}) &= \text{eq}_\Sigma(v, v') \quad \text{(otherwise)}
\end{aligned}$$

$\text{elems}(v)$	*sequence of values*

Elements of a list or tuple, or the characters of a string, in order.

$$\begin{aligned}
\text{elems}([\vec{v}]) &= \vec{v} \\
\text{elems}((\vec{v})) &= \vec{v} \\
\text{elems}(w) &= w_1 \cdot \ldots \cdot w_m && \text{where } w = w_1 \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} w_m, \text{ each } w_i \text{ one character}
\end{aligned}$$

$\text{iter}(v)$	*sequence of values*

Sequence drawn from v by a generator: the keys when v is a dictionary, the numbers $m, \ldots, n-1$ when v is a range with start m and stop n, and otherwise the elements of v.

$$\begin{aligned}
\text{iter}(\vec{w} : \vec{v}) &= \vec{w} \\
\text{iter}(\mathsf{Obj}(\texttt{builtins.range}, \texttt{start} : m \cdot \texttt{stop} : n)) &= m \cdot \ldots \cdot n-1 && \text{empty if } n \le m \\
\text{iter}(v) &= \text{elems}(v) && \text{(otherwise)}
\end{aligned}$$

$\text{getitem}(v, v')$	*result*

Subscripting: the value under key v' of a dictionary, or the element at index v' of a sequence, counting from the end when the index is negative. Aborts with KeyError or IndexError when Python raises the exception of that name. Aborts with excluded in three cases which Python allows but no typing rule for subscripts accepts: a range subscripted at all, a sequence indexed by a Boolean, which Python treats as the integer 0 or 1, and a dictionary subscripted by a key other than a string (Chapter 4). Aborts with TypeError for any other pair of

values.

$$
\begin{aligned}
\text{getitem}(\delta, w) &= \mathsf{val}(\delta(w)) && \text{if } w \in \text{dom}(\delta)\\
\text{getitem}(\delta, w) &= \mathsf{aborts}(\mathsf{KeyError}) && \text{if } w \notin \text{dom}(\delta)\\
\text{getitem}(\delta, v) &= \mathsf{aborts}(\mathsf{excluded}) && \text{if } v \text{ not a string}\\
\text{getitem}(v, n) &= \mathsf{val}(v_{n+1}) && \text{if elems}(v) = v_1 \cdot \ldots \cdot v_m,\ n \text{ integer and } 0 \le n < m\\
\text{getitem}(v, n) &= \mathsf{val}(v_{m+n+1}) && \text{if elems}(v) = v_1 \cdot \ldots \cdot v_m,\ n \text{ integer and } -m \le n < 0\\
\text{getitem}(v, n) &= \mathsf{aborts}(\mathsf{IndexError}) && \text{if elems}(v) = v_1 \cdot \ldots \cdot v_m,\ n \text{ integer and } (n < -m \text{ or } n \ge m)\\
\text{getitem}(v, v') &= \mathsf{aborts}(\mathsf{excluded}) && \text{if } v \text{ a list, tuple or string and } v' \text{ a Boolean}\\
\text{getitem}(v, v') &= \mathsf{aborts}(\mathsf{excluded}) && \text{if } v \text{ a range}\\
\text{getitem}(v, v') &= \mathsf{aborts}(\mathsf{TypeError}) && \text{otherwise}
\end{aligned}
$$

result(o) *result*

Result of a call from the outcome of its body: the value returned, or `None` if the body completes without returning, or the abort if the body aborts. The body prints nothing, since print statements are typed only outside function bodies (print).

$$
\begin{aligned}
\text{result}((\varepsilon, \mathsf{returns}(v))) &= \mathsf{val}(v)\\
\text{result}((\varepsilon, \mathsf{assigns}(\rho))) &= \mathsf{val}(\texttt{None})\\
\text{result}((\varepsilon, \mathsf{aborts}(\kappa))) &= \mathsf{aborts}(\kappa)
\end{aligned}
$$

$\text{match}_\Sigma(\rho, p, v)$ *match result*

Result of matching pattern p against value v under environment ρ: the bindings of the pattern variables if p matches v, and no-match if it does not. The result is undef for a numeric literal pattern if the equality of the literal and the value is undefined. The result is excluded for an int literal pattern against a float, a list pattern against a tuple, a tuple pattern against a list, or either against a range, which Python matches. A constructor pattern against an object is matched by match-fields$_\Sigma$ (§A.4). Before the value is examined, the head of a constructor pattern is resolved as Python resolves it: NameError if its first name is unbound, AttributeError if a later name is not found, excluded if it denotes a predefined class of `builtins`, which Python matches by an instance test, and TypeError if it denotes any other value which is not a class. A Boolean literal pattern does not match a number, since Python compares the value with the literal by identity. Sub-patterns are matched left to right, their results composing with $\otimes$ (§4.4).

$$
\begin{aligned}
\text{match}_\Sigma(\rho, n, v) &= \mathsf{excluded} && \text{if } n \text{ an int literal and } v \text{ a float}\\
\text{match}_\Sigma(\rho, \ell, v) &= \mathsf{match}(\emptyset) && \text{if } \text{eq}_\Sigma(\ell, v) = \mathsf{val}(\texttt{True})\\
\text{match}_\Sigma(\rho, \ell, v) &= \mathsf{no\text{-}match} && \text{if } \text{eq}_\Sigma(\ell, v) = \mathsf{val}(\texttt{False})\\
\text{match}_\Sigma(\rho, \texttt{True}, n) &= \mathsf{no\text{-}match}\\
\text{match}_\Sigma(\rho, \texttt{False}, n) &= \mathsf{no\text{-}match}\\
\text{match}_\Sigma(\rho, \ell, v) &= \mathsf{undef} && \text{if } \text{eq}_\Sigma(\ell, v) = \mathsf{aborts}(\mathsf{undef}) \text{ and } \ell \text{ a number}\\
\text{match}_\Sigma(\rho, x, v) &= \mathsf{match}(\{x : v\})\\
\text{match}_\Sigma(\rho, _, v) &= \mathsf{match}(\emptyset)\\
\text{match}_\Sigma(\rho, p \texttt{ as } x, v) &= \text{match}_\Sigma(\rho, p, v) \otimes \mathsf{match}(\{x : v\})\\
\text{match}_\Sigma(\rho, [p_1, \ldots, p_n], [v_1, \ldots, v_n]) &= \text{match}_\Sigma(\rho, p_1, v_1) \otimes \ldots \otimes \text{match}_\Sigma(\rho, p_n, v_n)\\
\text{match}_\Sigma(\rho, [\vec{p}], (\vec{v})) &= \mathsf{excluded}\\
\text{match}_\Sigma(\rho, (p_1, \ldots, p_n), (v_1, \ldots, v_n)) &= \text{match}_\Sigma(\rho, p_1, v_1) \otimes \ldots \otimes \text{match}_\Sigma(\rho, p_n, v_n)
\end{aligned}
$$

$$
\begin{aligned}
\text{match}_\Sigma(\rho, (\vec{p}), [\vec{v}]) &= \text{excluded} \\
\text{match}_\Sigma(\rho, [\vec{p}], \text{Obj}(\texttt{builtins.range}, \vec{x} : \vec{v})) &= \text{excluded} \\
\text{match}_\Sigma(\rho, (\vec{p}), \text{Obj}(\texttt{builtins.range}, \vec{x} : \vec{v})) &= \text{excluded} \\
\text{match}_\Sigma(\rho, \{w_1 : p_1, \ldots, w_n : p_n\}, \delta) &= \text{match}_\Sigma(\rho, p_1, \delta(w_1)) \otimes \ldots \otimes \text{match}_\Sigma(\rho, p_n, \delta(w_n)) \\
&\qquad \text{if } w_i \in \text{dom}(\delta) \text{ for each } i \\
\text{match}_\Sigma(\rho, (x \cdot \vec{x})(a), v) &= \text{NameError} \quad \text{if } \rho(x) \in \{\bot, \text{Unbound}\} \\
\text{match}_\Sigma(\rho, (x \cdot \vec{x})(a), v) &= \text{AttributeError} \\
&\qquad \text{if resolve}(\rho, x \cdot \vec{x}) \text{ undefined and } \rho(x) \notin \{\bot, \text{Unbound}\} \\
\text{match}_\Sigma(\rho, q(a), v) &= \text{excluded} \\
&\qquad \text{if resolve}(\rho, q) \text{ of the form } \text{Predef}(\texttt{builtins}.y) \\
\text{match}_\Sigma(\rho, q(a), v) &= \text{TypeError} \\
&\qquad \text{if resolve}(\rho, q) \text{ defined and not of the form} \\
&\qquad \text{Type}(C) \text{ or } \text{Predef}(\texttt{builtins}.y) \\
\text{match}_\Sigma(\rho, q(a), \text{Obj}(D, \vec{z} : \vec{v})) &= \text{match-fields}_\Sigma(\rho, C, a, \vec{z} : \vec{v}) \\
&\qquad \text{if resolve}(\rho, q) = \text{Type}(C) \text{ and } C \in \text{ancestors}_\Sigma(D) \\
\text{match}_\Sigma(\rho, p, v) &= \text{no-match} \quad \text{(otherwise)}
\end{aligned}
$$

$\text{match-fields}_\Sigma(\rho, C, a, \vec{z} : \vec{v})$	*match result*

Result of matching the sub-patterns of a against the fields $\vec{z} : \vec{v}$ of an object whose class has C among its ancestors: the sub-patterns match the fields of C named by $\text{field-map}_\Sigma(C, a)$, composing with $\otimes$; TypeError if a has more positional sub-patterns than C has fields, or two sub-patterns for one field, as in Python; and excluded if a otherwise does not cover the fields of C exactly, which Python allows. For the class `builtins.range`, for which Python does not support positional sub-patterns, a positional sub-pattern gives TypeError, and the keyword form, which Python matches, gives excluded.

$$
\begin{aligned}
\text{match-fields}_\Sigma(\rho, C, a, \vec{z} : \vec{v}) &= \text{match}_\Sigma(\rho, p_1, v_{j_1}) \otimes \ldots \otimes \text{match}_\Sigma(\rho, p_n, v_{j_n}) \\
&\qquad \text{if field-map}_\Sigma(C, a) = \{x_1 : p_1, \ldots, x_n : p_n\} \\
&\qquad \text{and } x_i = z_{j_i} \text{ for each } i, \text{ and } C \neq \texttt{builtins.range} \\
\text{match-fields}_\Sigma(\rho, C, (\vec{p}, \vec{y}{=}\vec{p}\,'), \vec{z} : \vec{v}) &= \text{TypeError} \\
&\qquad \text{if } C = \texttt{builtins.range} \text{ and } |\vec{p}| \geq 1 \\
\text{match-fields}_\Sigma(\rho, C, (\vec{p}, \vec{y}{=}\vec{p}\,'), \vec{z} : \vec{v}) &= \text{TypeError} \\
&\qquad \text{if fields}_\Sigma(C) = \forall \vec{\alpha}.\, \vec{x} : \vec{\sigma}, \text{ and either } |\vec{p}| > |\vec{x}| \\
&\qquad \text{or } y_i = x_j \text{ for some } i \text{ and some } j \leq |\vec{p}| \\
\text{match-fields}_\Sigma(\rho, C, a, \vec{z} : \vec{v}) &= \text{excluded} \quad \text{(otherwise)}
\end{aligned}
$$

$\text{dispatch}(\rho, v, \vec{p}, \vec{s})$	*bindings and statement, or termination kind*

Case taken by a `match` statement for value v: the bindings and statement of the first pattern that matches, or no bindings and `pass` if no pattern matches, and the termination kind κ if matching aborts with κ before any pattern matches.

$$
\begin{aligned}
\text{dispatch}(\rho, v, \varepsilon, \varepsilon) &= (\emptyset, \texttt{pass}) \\
\text{dispatch}(\rho, v, p \cdot \vec{p}, s \cdot \vec{s}) &= (\rho', s) && \text{if match}_\Sigma(\rho, p, v) = \text{match}(\rho') \\
\text{dispatch}(\rho, v, p \cdot \vec{p}, s \cdot \vec{s}) &= \text{dispatch}(\rho, v, \vec{p}, \vec{s}) && \text{if match}_\Sigma(\rho, p, v) = \text{no-match} \\
\text{dispatch}(\rho, v, p \cdot \vec{p}, s \cdot \vec{s}) &= \kappa && \text{if match}_\Sigma(\rho, p, v) = \kappa
\end{aligned}
$$

update(δ, δ') *dictionary entries*

Dictionary entries δ updated by the bindings of δ' in order, each replacing any earlier entry under the same key.

$$\begin{aligned} \text{update}(\delta, \varepsilon) &= \delta \\ \text{update}(\delta, w : v \cdot \delta') &= \text{update}(\text{setitem}(\delta, w, v), \delta') \end{aligned}$$

setitem(δ, w, v) *dictionary entries*

Dictionary entries δ with w bound to v, replacing any earlier binding of w.

$$\begin{aligned} \text{setitem}(\varepsilon, w, v) &= w : v \\ \text{setitem}(w : v' \cdot \delta, w, v) &= w : v \cdot \delta \\ \text{setitem}(w' : v' \cdot \delta, w, v) &= w' : v' \cdot \text{setitem}(\delta, w, v) && \text{if } w \neq w' \end{aligned}$$

Annex B

Excluded Python behaviours (informative)

This annex is informative and imposes no requirements; the exclusions which it describes follow from the syntax, well-formedness rules or semantics of PurePy, and their classification follows from §1.1. It summarises the Python behaviours excluded by PurePy by design (§B.1) with the stage at which each exclusion takes effect, lists the common Python features omitted from this version (§B.2), and lists the Python runtime errors which well-formedness prevents (§B.3).

B.1 Behaviours excluded by design

B.1.1 Soft keywords as identifiers

Python's soft keywords `match`, `case`, `type` and _ are keywords only in `match`, `case` and `type` statements and wildcard patterns, and identifiers elsewhere. In PurePy they are not identifiers at all (§2.1), so that whether a word is a keyword does not depend on where it occurs.

B.1.2 Mutable state

The excluded Python features associated with mutation and shared state are listed in Figure 1.1–1. Figure B.1–1 lists the other Python behaviours associated with mutation, and says how and when PurePy excludes each.

Construct	Python	Exclusion	Mechanism
Reassignment	Changes value of variable	Static	Variable declared once and assigned at most once on any path
Lambdas capturing comprehension variable	Variable mutated on each iteration; lambda sees final value	Static	Lambda may not capture generator variable

Figure B.1–1: Python mutation of state excluded by PurePy

B.1.3 Implicit coercions

Python accepts several implicit coercions and lax matches. Figure B.1–2 lists those excluded by PurePy and records at which stage each exclusion takes effect. The figure also lists operations with an outcome which is not specified by the Python language reference and which differs between implementations. A program which reaches a dynamically excluded operation runs under Python but has undefined behaviour under PurePy. A run which reaches a statically excluded operation aborts with excluded (Chapter 4); such runs are ruled out by type soundness (§4.6).

B.1.4 Load-dependent meaning

A name's meaning in PurePy must not depend on the order in which modules load or on runtime state. The Python features excluded for this reason are listed in Figure 1.1–1. Figure B.1–3 lists the other import and module behaviours of Python excluded for this reason, and says how and when PurePy excludes each.

The last three exclusions rule out programs whose meaning under Python changes when a module elsewhere in the program is loaded. A submodule is not an attribute of its parent until it is loaded, so after `import pkg`

Construct	Python	Exclusion	Mechanism
Non-Boolean test of `if` or `assert`, conditional expression, comprehension guard, first operand of `and` or `or`, or operand of `not`	Coerced to `bool`	Static	Test checked against `bool` (if, assert, cond, qual-guard, and, or, unop)
Boolean operand of arithmetic or ordering, index of a sequence, or argument of `round`	Treated as the integer 0 or 1	Static	`bool` not a subtype of `int` (binop, unop, subscript-list, subscript-tuple-int, subscript-str, call)
Argument of `sys.exit` other than an integer	Exits with status 0 for `None`; otherwise prints the argument and exits with status 1	Static	Argument checked against `int` (call)
List pattern at tuple, or tuple pattern at list	Matches, since list and tuple patterns both match lists and tuples	Static	Pattern fails at a type with tuple or list values (check-list-none, check-tuple-none)
List or tuple pattern at `range`	Matches, since `range` is a sequence	Static	Pattern fails at a type with range values (check-list-none, check-tuple-none)
Int literal pattern at `float`	Matches an equal float, so `1` matches `1.0`	Static	Pattern fails at a type with float values (check-literal)
Positional constructor pattern with fewer sub-patterns than fields	Matches, ignoring remaining fields	Static	Arguments cover fields exactly (field-map$_\Sigma$)
Non-string dictionary key	Any hashable value accepted, with `1`, `1.0` and `True` equivalent as keys	Syntactic in patterns, static in expressions	Keys are string literals in patterns, of type `str` in expressions (dict, dict-comp, subscript-dict and the `in` overload at `dict`)
Non-string `assert` message	Any object accepted, shown in traceback	Static	Message checked against `str` (assert-msg)
`==` or `!=` between operands of unrelated types	Defined, giving `False`	Static	Operand types must be comparable (binop)
Boolean against number under `==` or `!=`	Compared as numbers, so `True` equals `1`	Static; dynamic at `object`	`bool` and `int` incomparable (binop); at `object`, equality undefined
Function under `==` or `!=`	Compared by identity, so two textually identical lambdas are unequal	Static; dynamic at `object`	Callable types not equality types (binop); at `object`, undefined between functions
`nan` under container equality or membership	Elements compared by identity before equality, so identical NaNs match but distinct ones do not	Dynamic	Undefined when both elements are `nan`
Overflow of `**` on floats	`OverflowError` in CPython, an infinity in GraalPy	Dynamic	Undefined when result rounds to infinity

Figure B.1–2: Python coercions, lax matches and non-portable operations excluded by PurePy

Construct	Python	Exclusion	Mechanism
Submodule importing itself from parent, `from a import b` in module `a.b`	Parent may still be loading	Static	From-imported submodule is a dependency, so the import is a cycle
Import of contained module, `import x.y` in module `x`	Binding for head denotes still-loading module	Static	Modules which strictly contain the target of a plain import are checked, importing module included
Reference `pkg.sub.y` after `import pkg` alone	Resolves if any module has loaded `pkg.sub`; `AttributeError` otherwise	Static	Qualified names resolve through imports of referring module
Reference to module member assigned on some paths only	`AttributeError` if path not taken	Static	Member must be definitely assigned
Module binding name of one of its submodules	Attribute set by assignment and by loading submodule; last wins	Static	Names bound by module disjoint from names of its submodules

Figure B.1–3: Python import and module behaviours excluded by PurePy

alone, Python resolves `pkg.sub.y` if some earlier import, anywhere in the program, has loaded `pkg.sub`, and raises `AttributeError` otherwise. An attribute of a module can be set both by an assignment in the module body and by loading a same-named submodule, each overwriting the other, so the value of the attribute depends on load order. Consider a program with four modules:

```
# pkg                 # pkg.sub             # helper                  # __main__
sub: int = 1          # (empty)             import pkg.sub            import pkg
                                                                      print(pkg.sub) # 1
```

Here `helper` is never imported, so `pkg.sub` never loads, and `pkg.sub` in the main module denotes the assignment. Now let the main module also import `helper`, which it does not otherwise use:

```
# __main__
import helper
import pkg
print(pkg.sub) # <module 'pkg.sub'>
```

Loading `pkg.sub`, wherever in the program it happens, sets it as an attribute of the shared `pkg` object, so the extra import changes the meaning of `pkg.sub` in the main module, even though it does not import `pkg.sub`. PurePy rejects `pkg`, because it binds the name of its submodule.

B.1.5 Compatibility with mypy

Some rules reject programs that could be accepted without loss of soundness, only so that a well-typed program is also well typed under mypy (Chapter 3). Figure B.1–4 lists each restriction with mypy's verdict and the rule that enforces it. Figure 1.2–1 includes bugs reported in mypy which make it reject some well-formed programs.

Construct	mypy	Mechanism
Call returning `None` as an expression	Rejects any use of the value	call requires a result other than `None`; the call is a statement (call-stmt)
Body that may fall through at a return type such as `int | None`	Reports a missing return	def-body requires outcome Returns unless the return type is equivalent to `None`
Assertion of expression of type `Literal[False]` other than `False`, such as `(False,)[0]`	Types `(False,)[0]` at `bool`, so reports a missing return	assert-false applies to the literal `False` only
Bare `return` at such a return type	Reports a missing return value	return-none requires the return type equivalent to `None`
Match whose cases cover a class field by field	Narrows the scrutinee by whole classes, so reports a missing return	Exhaustiveness decided by the remaining type (match)
Pattern variable after a case excluding a field value	Binds at the field type	Bindings taken from the remaining type (match-case, check-var)
`list[bool]` below `list[int]`	Rejects; list and dictionary types invariant	subty-list and subty-dict require equivalent components
Declaration or import of a name already a parameter, imported name, pattern variable or declaration of the scope	Rejects the redeclaration	Context scope($\vec{y}$, s) of the body undefined unless the names are pairwise distinct; a declaration alone requires only that its name be unbound (declare)

Figure B.1–4: Restrictions imposed for compatibility with mypy

B.2 Features omitted from this version

Figure B.2–1 lists common Python features omitted from this version (§1.1).

B.3 Runtime errors prevented by well-formedness

Well-formedness prevents the Python runtime errors listed in Figure B.3–1 as well as the statically excluded behaviours of Figure B.1–2.

Feature	Examples
Sets	Set literals, set comprehensions, `set` and `frozenset`
Operators	Bitwise operators `&`, `|`, `^`, `~`, `<<` and `>>`; chained comparisons `a < b < c`; `is` and `is not` with `None`, `True` or `False`
Expressions	Slicing, f-strings, starred expressions, dictionary unpacking
Literals	Complex, bytes and ellipsis literals
Assignment	Unpacking assignment `a, b = t`, chained assignment `a = b = 1`, unpacking in comprehensions
Parameters and arguments	Default values, `*args`, `**kwargs`, keyword-only and positional-only parameters, unpacking in argument lists, `print` with several arguments or keyword arguments
Function definitions	Decorators
Classes	Classes other than dataclasses, methods, class decorators other than `@dataclass`, field defaults, multiple inheritance, class keyword arguments, enumerations, classes defined outside module top level, classes as values
Patterns	Case guards, or-patterns, star patterns, attribute value patterns, complex and bytes literal patterns, rest capture in dictionary patterns
Types	Bounds and constraints on type parameters, `ParamSpec`, `TypeVarTuple`, type statements outside module top level, `Callable[..., int]`, `tuple[int, ...]`, `Literal` with several values, string annotations
Imports	`import a as b`, `from a import b as c`, `import a, b`, relative imports
Exceptions and effects	`try`, `except`, `finally`, `raise`, `with`, `async`, `await`, `input`, `open`, expression-level printing
Library	Standard library modules other than `builtins`, `dataclasses`, `sys` and `typing`, such as `re`; attributes of those modules not listed in Figure 2.4–1, such as `sum` and `sys.version`; attributes of built-in types other than mutating methods, such as `str.upper`

Figure B.2–1: Common Python features omitted from this version

Python error	Mechanism	See
`NameError`, `UnboundLocalError`	Variable use permitted only where definitely assigned; fields and base class required to name earlier type definitions	Figure 3.3–1, Figure 3.2–1, Figure 3.3–5
`ModuleNotFoundError`	Import target required to be a program module	Figure 3.5–2a
`ImportError`	Each imported name required to be a member of the source module; import cycles rejected, where Python may run or fail depending on entry point	Figure 3.5–1, Figure 3.5–3
`AttributeError`	Module attribute required to be definitely assigned; object attribute required to be a field of the class	Figure 3.2–1
`TypeError`	Operator overload required at operand types	Figure 3.2–1
	Callee required to be a function or class, with arguments matching parameters or fields	Figure 3.2–1
	`range` rejected as base class	Figure 3.3–5
	Constructor pattern required to match class arity; Python permits fewer sub-patterns	Figure 3.4–4a

Figure B.3–1: Python errors prevented by well-formedness

Bibliography

Apple Inc. *The Swift Programming Language*. Apple, 2025. URL https://docs.swift.org/swift-book/documentation/the-swift-programming-language/.

Rowan Davies and Frank Pfenning. Intersection Types and Computational Effects. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 198–208, 2000. doi: 10.1145/351240.351259.

Jana Dunfield and Neel Krishnaswami. Bidirectional Typing. *ACM Computing Surveys*, 54(5):1–38, 2021. doi: 10.1145/3450952.

Jana Dunfield and Frank Pfenning. Tridirectional Typechecking. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '04)*, pages 281–292, 2004. doi: 10.1145/964001.964025.

James Gosling, Bill Joy, Guy L. Steele Jr., Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. *The Java Language Specification, Java SE 24 Edition*. Oracle, 2025. URL https://docs.oracle.com/javase/specs/jls/se24/html/index.html.

Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001. ISSN 0164-0925, 1558-4593. doi: 10.1145/503502.503505.

JetBrains. *Kotlin Language Documentation*. JetBrains, 2025. URL https://kotlinlang.org/docs/home.html.

Georgios Karachalias, Tom Schrijvers, Dimitrios Vytiniotis, and Simon Peyton Jones. GADTs Meet Their Match: Pattern-Matching Warnings That Account for GADTs, Guards, and Laziness. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, pages 424–436. ACM, 2015. doi: 10.1145/2784731.2784748.

Luc Maranget. Warnings for Pattern Matching. *Journal of Functional Programming*, 17(3):387–421, 2007. doi: 10.1017/S0956796807006223.

Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Sean McDirmid, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Lex Spoon, and Matthias Zenger. *The Scala Language Specification, Version 2.13*. EPFL, 2025. URL https://scala-lang.org/files/archive/spec/2.13/.

Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

Python Typing Council. Specification for the Python Type System. Python typing documentation, 2025. URL https://typing.python.org/en/latest/spec/.

Peter Sestoft. ML Pattern Match Compilation and Partial Evaluation. In *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 446–464. Springer, 1996. doi: 10.1007/3-540-61580-6_23.

W. G. van Hoorn and B. van Rootselaar. Fundamental Notions in the Theory of Seminearrings. *Compositio Mathematica*, 18(1-2):65–78, 1967. URL https://www.numdam.org/item/CM_1967__18_1-2_65_0/.