Python: The Full Monty
A Tested Semantics for the Python Programming Language

Joe Gibbs Politz
Providence, RI, USA
joe@cs.brown.edu

Alejandro Martinez
La Plata, BA, Argentina
amtriathlon@gmail.com

Matthew Milano
Providence, RI, USA
matthew@cs.brown.edu

Sumner Warren
Providence, RI, USA
jswarren@cs.brown.edu

Daniel Patterson
Providence, RI, USA
dbpatter@cs.brown.edu

Junsong Li
Beijing, China
ljs.darkfish@gmail.com

Anand Chitipothu
Bangalore, India
anandology@gmail.com

Shriram Krishnamurthi
Providence, RI, USA
sk@cs.brown.edu

Abstract

We present a small-step operational semantics for the Python programming language. We present both a core language for Python, suitable for tools and proofs, and a translation process for converting Python source to this core. We have tested the composition of translation and evaluation of the core for conformance with the primary Python implementation, thereby giving confidence in the fidelity of the semantics. We briefly report on the engineering of these components. Finally, we examine subtle aspects of the language, identifying scope as a pervasive concern that even impacts features that might be considered orthogonal.

Categories and Subject Descriptors J.3 [*Life and Medical Sciences*]: Biology and Genetics

Keywords serpents

1. Motivation and Contributions

The Python programming language is currently widely used in industry, science, and education. Because of its popularity it now has several third-party tools, including analyzers that check for various potential error patterns [2, 5, 11, 13]. It also features interactive development environments [1, 8, 14] that offer a variety of features such as variable-renaming refactorings and code completion. Unfortunately, these tools are unsound: for instance, the simple eight-line program shown in the appendix uses no "dynamic" features and confuses the variable renaming feature of these environments.

The difficulty of reasoning about Python becomes even more pressing as the language is adopted in increasingly important domains. For instance, the US Securities and Exchange Commission has proposed using Python as an executable specification of financial contracts [12], and it is now being used to script new network paradigms [10]. Thus, it is vital to have a precise semantics available for analyzing programs and proving properties about them.

This paper presents a semantics for much of Python (section 5). To make the semantics tractable for tools and proofs, we divide it into two parts: a core language, λ_π, with a small number of constructs, and a desugaring function that translates source programs into the core.[1] The core language is a mostly traditional stateful lambda-calculus augmented with features to represent the essence of Python (such as method lookup order and primitive lists), and should thus be familiar to its potential users.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
OOPSLA '13, October 29-31 2013, Indianapolis, IN, USA.
Copyright © 2013 ACM 978-1-4503-2374-1/13/10…$15.00.
http://dx.doi.org/10.1145/2509136.2509536

[1] The term *desugaring* is evocative but slightly misleading, because ours is really a compiler to a slightly different language. Nevertheless, it is more suggestive than a general term like "compiler". We blame Arjun Guha for the confusing terminology.

Because desugaring converts Python surface syntax to the core language, when it is composed with an interpreter for λ_π (which is easy to write), we have another implementation of Python. We can then ask how this implementation compares with the traditional CPython implementation, which represents a form of ground truth. By carefully adjusting the core and desugaring, we have achieved high fidelity with CPython. Therefore, users can built tools atop λ_π, confident that they are conformant with the actual language.

In the course of creating this high-fidelity semantics, we identified some peculiar corners of the language. In particular, scope is non-trivial and interacts with perhaps unexpected features. Our exposition focuses on these aspects.

In sum, this paper makes the following contributions:

- a *core semantics* for Python, dubbed λ_π, which is defined as a reduction semantics using PLT Redex [3];
- an *interpreter*, dubbed $\lambda_{\pi\downarrow}$, implemented in 700LOC of Racket, that has been tested against the Redex model;
- a *desugaring* translation from Python programs to λ_π, implemented in Racket;
- a demonstration of *conformance* of the composition of desugaring with $\lambda_{\pi\downarrow}$ to CPython; and,
- *insights* about Python gained from this process.

Presenting the semantics in full is neither feasible, given space limits, nor especially enlightening. We instead focus on the parts that are important or interesting. We first give an overview of λ_π's value and object model. We then introduce desugaring through classes. We then discuss generators, classes, and their interaction with scope. Finally, we describe the results of testing our semantics against CPython. All of our code is available online at `https://www.github.com/brownplt/lambda-py`.

2. Warmup: A Quick Tour of λ_π

We provide an overview of the object model of λ_π and Python, some of the basic operations on objects, and the shape of our small step semantics. This introduces notation and concepts that will be used later to explain the harder parts of Python's semantics.

2.1 λ_π Values

Figure 1 shows all the values and expressions of λ_π. The metavariables *v* and *val* range over the values of the language. All values in λ_π are either objects, written as triples in ⟨ ⟩, or references to entries in the store Σ, written `@`*ref*.

Each λ_π object is written as a triple of one of the forms:

⟨*v*, *mval*, {*string*: *ref*, ...}⟩
⟨*x*, *mval*, {*string*: *ref*, ...}⟩

These objects have their *class* in the first position, their primitive content in the second, and the dictionary of string-indexed fields that they hold in the third. The class value is either another λ_π value or the name of a built-in class. The primitive content, or *meta-val*, position holds special kinds of builtin data, of which there is one per builtin type that λ_π models: numbers, strings, the distinguished `meta-none` value, lists, tuples, sets, classes, and functions.[2]

```
        Σ ::= ((ref v+undef) ...)
      ref ::= natural
   v, val ::= ⟨val,mval,{string:ref,...}⟩
            | ⟨x,mval,{string:ref,...}⟩
            | @ref | (sym string)
  v+undef ::= v | ☠
  e+undef ::= e | ☠
        t ::= global | local
     mval ::= (no-meta) | number | string | meta-none
            | [val ...] | (val ...) | {val ...}
            | (meta-class x) | (meta-code (x ...) x e)
            | λ(x ...) opt-var.e
  opt-var ::= (x) | (no-var)
        e ::= v | ref | (fetch e) | (set! e e) | (alloc e)
            | e[e] | e[e := e]
            | if e e e | e e
            | let x = e+undef in e
            | x | e := e | (delete e)
            | e (e ...) | e (e ...)*e | (frame e) | (return e)
            | (while e e e) | (loop e e) | break | continue
            | (builtin-prim op (e ...))
            | fun (x ...) opt-var e
            | ⟨e,mval⟩ | list⟨e,[e ...]⟩
            | tuple⟨e,(e ...)⟩ | set⟨e,(e ...)⟩
            | (tryexcept e x e e) | (tryfinally e e)
            | (raise e) | (err val)
            | (module e e) | (construct-module e)
            | (in-module e ε)
```

Figure 1: λ_π expressions

The distinguished ☠ ("skull") form represents uninitialized heap locations whose lookup should raise an exception. Within expressions, this form can *only* appear in `let`-bindings whose binding position can contain both expressions and ☠. The evaluation of ☠ in a `let`-binding allocates it on the heap. Thereafter, it is an error to look up a store location containing ☠; that location must have a value *val* assigned into it for lookup to succeed. Figure 2 shows the behavior of lookup in heaps Σ for values and for ☠. This notion of undefined locations will come into play when we discuss scope in section 4.2.

Python programs cannot manipulate object values directly; rather, they always work with references to objects. Thus, many of the operations in λ_π involve the heap, and few are purely functional. As an example of what such an operation looks like, take constructing a list. This takes the

[2] We express dictionaries in terms of lists and tuples, so we do not need to introduce a special *mval* form for them.

```
    (E[ let  x = v+undef  in e ] ε Σ)                 [E-LetLocal]
──→ (E[[x/ref]e ] ε Σ1)
     where (Σ1 ref) = alloc(Σ,v+undef)

    (E[ref] ε Σ) ──→ (E[val] ε Σ)                        [E-GetVar]
     where Σ = ((ref1 v+undef1) ... (ref val) (refn v+undefn) ...)

    (E[ref] ε Σ)                                    [E-GetVarUndef]
──→ (E[(raise ⟨%str,"Uninitialized local",{}⟩)] ε Σ)
     where Σ = ((ref1 v+undef1) ... (ref ☠) (refn v+undefn) ...)
```

Figure 2: Let-binding identifiers, and looking up references

```
    (E[⟨val,mval⟩] ε Σ)                       [E-Object]
──→ (E[@ref_new ] ε Σ1)
     where (Σ1 ref_new) = alloc(Σ,⟨val,mval,{}⟩)

    (E[ tuple⟨valc,(val ...)⟩] ε Σ)           [E-Tuple]
──→ (E[@ref_new ] ε Σ1)
     where (Σ1 ref_new) = alloc(Σ,⟨valc,(val ...),{}⟩)

    (E[ set⟨valc,(val ...)⟩] ε Σ)             [E-Set]
──→ (E[@ref_new ] ε Σ1)
     where (Σ1 ref_new) = alloc(Σ,⟨valc,{val ...},{}⟩)
```

Figure 3: λ_π reduction rules for creating objects

values that should populate the list, store them in the heap, and return a pointer to the newly-created reference:

```
    (E[ list⟨valc,[val ...]⟩] ε Σ)            [E-List]
──→ (E[@ref_new ] ε Σ1)
     where (Σ1 ref_new) = alloc(Σ,⟨valc,[val ...],{}⟩)
```

E-List is a good example for understanding the shape of evaluation in λ_π. The general form of the reduction relation is over expressions *e*, global environments *ε*, and heaps *Σ*:

$$(e\ \varepsilon\ \Sigma) \longrightarrow (e\ \varepsilon\ \Sigma)$$

In the E-List rule, we also use evaluation contexts *E* to enforce an order of operations and eager calling semantics. This is a standard application of Felleisen-Hieb-style small-step semantics [4]. Saliently, a new list value is populated from the list expression via the `alloc` metafunction, this is allocated in the store, and the resulting value of the expression is a pointer ref_{new} to that new list.

Similar rules for objects in general, tuples, and sets are shown in figure 3. Lists, tuples, and sets are given their own expression forms because they need to evaluate their subexpressions and have corresponding evaluation contexts.

```
    (E[(fetch @ref )] ε Σ)      [E-Fetch]
──→ (E[Σ(ref)] ε Σ)

    (E[(set! @ref  val)] ε Σ)    [E-Set!]
──→ (E[val] ε Σ1)
     where Σ1 = Σ[ref:=val]

    (E[(alloc val)] ε Σ)        [E-Alloc]
──→ (E[@ref_new ] ε Σ1)
     where (Σ1 ref_new) = alloc(Σ,val)
```

Figure 5: λ_π reduction rules for references

2.2 Accessing Built-in Values

Now that we've created a list, we should be able to perform some operations on it, like look up its elements. λ_π defines a number of builtin primitives that model Python's internal operators for manipulating data, and these are used to access the contents of a given object's *mval*. We formalize these builtin primitives in a metafunction δ. A few selected cases of the δ function are shown in figure 4. This metafunction lets us, for example, look up values on builtin lists:

```
(prim "list-getitem" (⟨%list,[⟨%str,"first-elt",{}⟩],{}⟩
                      ⟨%int,0,{}⟩))

              ==> ⟨%str,"first-elt",{}⟩
```

Since δ works over object values themselves, and not over references, we need a way to access the values in the store. λ_π has the usual set of operators for accessing and updating mutable references, shown in figure 5. Thus, the real λ_π program corresponding to the one above would be:

```
(prim "list-getitem"
      ((fetch  list⟨%list,[⟨%str,"first-elt",{}⟩]⟩)
       (fetch ⟨%int,0⟩)))
```

Similarly, we can use `set!` and `alloc` to update the values in lists, and to allocate the return values of primitive operations.

$$\delta(\text{“}\texttt{list-getitem}\text{”}\ \langle any_{c1},\texttt{[}val_0\ \texttt{...}\ val_1\ val_2\ \texttt{...],}any_1\rangle\ \langle any_{c2},number_2,any_2\rangle) = val_1$$
$$\texttt{where (equal? (length (}val_0\ \texttt{...))}\ \ number_2\texttt{)}$$
$$\delta(\text{“}\texttt{list-getitem}\text{”}\ \langle any_{c1},\texttt{[}val_1\ \texttt{...],}any_1\rangle\ \langle any_{c2},number_2,any_2\rangle) = \langle\texttt{\%none, meta-none ,\{\}}\rangle$$
$$\delta(\text{“}\texttt{list-setitem}\text{”}\ \langle any_{c1},\texttt{[}val_0\ \texttt{...}\ val_1\ val_2\ \texttt{...],}any_1\rangle\ \langle x_2,number_2,any_2\rangle\ val_3\ val_4) = \langle val_4,\texttt{[}val_0\ \texttt{...}\ val_3\ val_2\ \texttt{...],\{\}}\rangle$$
$$\texttt{where (equal? (length (}val_0\ \texttt{...))}\ \ number_2\texttt{)}$$
$$\delta(\text{“}\texttt{num+}\text{”}\ \langle any_{cls},number_1,any_1\rangle\ \langle any_{cls2},number_2,any_2\rangle) = \langle any_{cls},\texttt{(+}\ number_1\ \ number_2\texttt{),\{\}}\rangle$$

Figure 4: A sample of λ_π primitives

We desugar to patterns like the above from Python's actual surface operators for accessing the elements of a list in expressions like `mylist[2]`.

2.3 Updating and Accessing Fields

So far, the dictionary part of λ_π objects have always been empty. Python does, however, support arbitrary field assignments on objects. The expression

$$e_{obj}\texttt{[}e_{str}\ \texttt{:=}\ e_{val}\texttt{]}$$

has one of two behaviors, defined in figure 6. Both behaviors work over references to objects, not over objects themselves, in contrast to δ. If e_{str} is a string object that is already a member of e_{obj}, that field is imperatively updated with e_{val}. If the string is not present, then a new field is added to the object, with a newly-allocated store position, and the object's location in the heap is updated.

The simplest rule for accessing fields simply checks in the object's dictionary for the provided name and returns the appropriate value, shown in E-GetField in figure 6. E-GetField also works over reference values, rather than objects directly.

2.4 First-class Functions

Functions in Python are objects like any other. They are defined with the keyword `def`, which produces a callable object with a mutable set of fields, whose class is the built-in `function` class. For example a programmer is free to write:

```
def f():
  return f.x

f.x = -1
f() # ==> -1
```

We model functions as just another kind of object value, with a type of *mval* that looks like the usual functional λ:

$$\lambda\texttt{(}x\ \texttt{...)}\ opt\text{-}var.e$$

The *opt-var* indicates whether the function is variable-arity: if *opt-var* is of the form (y), then if the function is called with more arguments than are in its list of variables $(x\ \ldots)$, they are allocated in a new tuple and bound to y in the body. Since these rules are relatively unexciting and verbose, we defer their explanation to the appendix.

2.5 Loops, Exceptions, and Modules

We defer a full explanation of the terms in figure 1, and the entire reduction relation, to the appendix. This includes a mostly-routine encoding of control operators via special evaluation contexts, and a mechanism for loading new code via modules. We continue here by focusing on cases in λ_π that are unique in Python.

3. Classes, Methods, and Desugaring

Python has a large system with first-class methods, implicit receiver binding, multiple inheritance, and more. In this section we discuss what parts of the class system we put in λ_π, and which parts we choose to eliminate by desugaring.

3.1 Field Lookup in Classes

In the last section, we touched on field lookup in an object's local dictionary, and didn't discuss the purpose of the class position at all. When an object lookup $\langle val_c, mval, \texttt{d}\rangle\texttt{[}e_{str}\texttt{]}$ doesn't find e_{str} in the local dictionary `d`, it defers to a lookup algorithm on the class value val_c. More specifically, it uses the "`__mro__`" (short for *method resolution order*) field of the class to determine which class dictionaries to search for the field. This field is visible to the Python programmer:

```
class C(object):
  pass # a class that does nothing

print(C.__mro__)
# (<class 'C'>, <class 'object'>)
```

Field lookups on objects whose class value is `C` will first look in the dictionary of `C`, and then in the dictionary of the built-in class `object`. We define this lookup algorithm within λ_π as `class-lookup`, shown in figure 7 along with the reduction rule for field access that uses it.

This rule allows us to model field lookups that defer to a superclass (or indeed, a list of them). But programmers don't explicitly define "`__mro__`" fields; rather, they use higher-level language constructs to build up the inheritance hierarchy the instances eventually use.

3.2 Desugaring Classes

Most Python programmers use the special `class` form to create classes in Python. However, `class` is merely syntac-

$$(E[@ref_{obj}\ [@ref_{str}\ := val_1]]\ \varepsilon\ \Sigma) \qquad \text{[E-SetFieldUpdate]}$$
$$\longrightarrow (E[val_1]\ \varepsilon\ \Sigma[ref_1 := val_1])$$
where $\langle any_{cls1}, mval, \{string_2 : ref_2, \ldots, string_1 : ref_1, string_3 : ref_3, \ldots\}\rangle = \Sigma(ref_{obj})$,
$\langle any_{cls2}, string_1, any_{dict}\rangle = \Sigma(ref_{str})$

$$(E[@ref_{obj}\ [@ref_{str}\ := val_1]]\ \varepsilon\ \Sigma) \qquad \text{[E-SetFieldAdd]}$$
$$\longrightarrow (E[val_1]\ \varepsilon\ \Sigma_2)$$
where $\langle any_{cls1}, mval, \{string : ref, \ldots\}\rangle = \Sigma(ref_{obj})$,
$(\Sigma_1\ ref_{new}) = \mathtt{alloc}(\Sigma, val_1)$,
$\langle any_{cls2}, string_1, any_{dict}\rangle = \Sigma(ref_{str})$,
$\Sigma_2 = \Sigma_1[ref_{obj} := \langle any_{cls1}, mval, \{string_1 : ref_{new}, string : ref, \ldots\}\rangle]$,
$(\mathtt{not}\ (\mathtt{member}\ string_1\ \ (string\ \ldots)))$

$$(E[@ref\ [@ref_{str}\ ]]\ \varepsilon\ \Sigma) \qquad \text{[E-GetField]}$$
$$\longrightarrow (E[\Sigma(ref_1)]\ \varepsilon\ \Sigma)$$
where $\langle any_{cls1}, string_1, any_{dict}\rangle = \Sigma(ref_{str})$,
$\langle any_{cls2}, mval, \{string_2 : ref_2, \ldots, string_1 : ref_1, string_3 : ref_3, \ldots\}\rangle = \Sigma(ref)$

Figure 6: Simple field access and update in λ_π

$$(E[@ref_{obj}\ [@ref_{str}\ ]]\ \varepsilon\ \Sigma) \qquad \text{[E-GetField-Class]}$$
$$\longrightarrow (E[val_{result}]\ \varepsilon\ \Sigma_{result})$$
where $\langle any_{cls}, string, any_{dict}\rangle = \Sigma(ref_{str})$,
$\langle @ref\ , mval, \{string_1 : ref_2, \ldots\}\rangle = \Sigma(ref_{obj})$,
$(\Sigma_{result}\ val_{result}) = \text{class-lookup}[\![@ref_{obj}\ , \Sigma(ref),\ string,\ \Sigma]\!]$,
$(\mathtt{not}\ (\mathtt{member}\ string\ \ (string_1\ \ldots)))$

$\text{class-lookup}[\![@ref_{obj}\ , \langle any_c, any_{mval}, \{string_1 : ref_1, \ldots, \text{“__mro__”} : ref, string_2 : ref_2, \ldots\}\rangle,\ string,\ \Sigma]\!] = (\Sigma\ val_{result})$
where $\langle any_1, (val_{cls}\ \ldots), any_3\rangle = \text{fetch-pointer}[\![\Sigma(ref),\ \Sigma]\!]$,
$val_{result} = \text{class-lookup-mro}[\![(val_{cls}\ \ldots),\ string,\ \Sigma]\!]$

$\text{class-lookup-mro}[\![(@ref_c\ \ val_{rest}\ \ldots),\ string,\ \Sigma]\!] = \Sigma(ref)$
where $\langle any_1, any_2, \{string_1 : ref_1, \ldots, string : ref, string_2 : ref_2, \ldots\}\rangle = \Sigma(ref_c)$

$\text{class-lookup-mro}[\![(@ref_c\ \ val_{rest}\ \ldots),\ string,\ \Sigma]\!] = \text{class-lookup-mro}[\![(val_{rest}\ \ldots),\ string,\ \Sigma]\!]$
where $\langle any_1, any_2, \{string_1 : ref_1, \ldots\}\rangle = \Sigma(ref_c)$, $(\mathtt{not}\ (\mathtt{member}\ string\ \ (string_1\ \ldots)))$

$\text{fetch-pointer}[\![@ref\ ,\ \Sigma]\!] = \Sigma(ref)$

Figure 7: Class lookup

tic sugar for a use of the builtin Python function `type`.[3] The documentation states explicitly that the two following forms [sic] produce *identical* type objects:

```
class X:
     a = 1

X = type('X', (object,), dict(a=1))
```

This means that to implement classes, we merely need to understand the built-in function `type`, and how it creates new classes on the fly. Then it is a simple matter to desugar class forms to this function call.

The implementation of `type` creates a new object value for the class, allocates it, sets the “`__mro__`” field to be the computed inheritance graph,[4] and sets the fields of the class to be the bindings in the dictionary. We elide some of the verbose detail in the iteration over `dict` by using the `for` syntactic abbreviation, which expands into the desired iteration:

[3] http://docs.python.org/3/library/functions.html#type

[4] This uses an algorithm that is implementable in pure Python: http://www.python.org/download/releases/2.3/mro/.

```
%type :=
fun (cls bases dict)
  let  newcls = (alloc 〈%type,(meta-class cls),{}〉)  in
   newcls[〈%str,"__mro__"〉 :=
   (builtin-prim "type-buildmro" (newcls bases))]
   (for (key elt) in dict[〈%str,"__items__"〉] ()
     newcls[key := elt])
   (return newcls)
```

This function, along with the built-in `type` class, suffices for bootstrapping the object system in λ_π.

3.3 Python Desugaring Patterns

Python objects can have a number of so-called *magic fields* that allow for overriding the behavior of built-in syntactic forms. These magic fields can be set anywhere in an object's inheritance hierarchy, and provide a lot of the flexibility for which Python is well-known.

For example, the field accesses that Python programmers write are not directly translated to the rules in λ_π. Even the execution of `o.x` depends heavily on its inheritance hierarchy. This program desugars to:

```
o[〈%str,"__getattribute__"〉] (o 〈%str,"x"〉)
```

For objects that don't override the "`__getattribute__`" field, the built-in object class's implementation does more than simply look up the "`x`" property using the field access rules we presented earlier. Python allows for attributes to implement special accessing functionality via *properties*,[5] which can cause special functions to be called on property access. The "`__getattribute__`" function of `object` checks if the value of the field it accesses has a special "`__get__`" method, and if it does, calls it:

```
object[〈%str,"__getattribute__"〉 :=
fun (obj field)
  let  value = obj[field]  in
    if (builtin-prim "has-field?" (value
                                   〈%str,"__get__"〉))
       (return value[〈%str,"__get__"〉] ())
       (return value)   ]
```

This pattern is used to implement a myriad of features. For example, when accessing function values on classes, the "`__get__`" method of the function binds the self argument:

```
class C(object):
  def f(self):
    return self

c = C() # constructs a new C instance
g = c.f # accessing c.f creates a
        # method object closed over c
g() is c # ==> True

# We can also bind self manually:
self_is_5 = C.f.__get__(5)
self_is_5() # ==> 5
```

[5] http://docs.python.org/3/library/functions.html#property

Thus, very few object-based primitives are needed to create static class methods and instance methods.

Python has a number of other special method names that can be overridden to provide specialized behavior. λ_π tracks Python this regard; it desugars surface expressions into calls to methods with particular names, and provides built-in implementations of those methods for arithmetic, dictionary access, and a number of other operations. Some examples:

`o[p]` *desugars to...* `o[〈%str,"__getitem__"〉] (p)`
`n + m` *desugars to...* `n[〈%str,"__add__"〉] (m)`
`fun(a)` *desugars to...* `fun[〈%str,"__call__"〉] (a)`

With the basics of `type` and object lookup in place, getting these operations right is just a matter of desugaring to the right method calls, and providing the right built-in versions for primitive values. This is the form of much of our desugaring, and though it is labor-intensive, it is also the straightforward part of the process.

4. Python: the Hard Parts

Not all of Python has a semantics as straightforward as that presented so far. Python has a unique notion of scope, with new scope operators added in Python 3 to provide some features of more traditional static scoping. It also has powerful control flow constructs, notably generators.

4.1 Generators

Python has a built-in notion of *generators*, which provide a control-flow construct, `yield`, that can implement lazy or generative sequences and coroutines. The programmer interface for creating a generator in Python is straightforward: any function definition that uses the `yield` keyword in its body is automatically converted into an object with a generator interface. To illustrate the easy transition from function to generator, consider this simple program:

```
def f():
  x = 0
  while True:
    x += 1
    return x

f() # ==> 1
f() # ==> 1
# ...
```

When called, this function always returns `1`.

Changing `return` to `yield` turns this into a generator. As a result, applying `f()` no longer immediately evaluates the body; instead, it creates an object whose `next` method evaluates the body until the next `yield` statement, stores its state for later resumption, and returns the yielded value:

```
def f():
  x = 0
  while True:
    x += 1
    yield x

gen = f()
gen.__next__() # ==> 1
gen.__next__() # ==> 2
gen.__next__() # ==> 3
# ...
```

Contrast this with the following program, which seems to perform a simple abstraction over the process of yielding:

```
def f():
  def do_yield(n):
    yield n
  x = 0
  while True:
    x += 1
    do_yield(x)
```

Invoking `f()` results in an *infinite loop*. That is because Python strictly converts only the innermost function with a `yield` into a generator, so only `do_yield` is a generator. Thus, the generator stores only the execution context of `do_yield`, not of `f`.

Failing to Desugar Generators with (Local) CPS

The experienced linguist will immediately see what is going on. Clearly, Python has made a design decision to store only *local* continuations. This design can be justified on the grounds that converting a whole program to continuation-passing style (CPS) can be onerous, is not modular, can impact performance, and depends on the presence of tail-calls (which Python does not have). In contrast, it is natural to envision a translation strategy that performs only a local conversion to CPS (or, equivalently, stores the local stack frames) while still presenting a continuation-free interface to the rest of the program.

From the perspective of our semantics, this is a potential boon: we don't need to use a CPS-style semantics for the whole language! Furthermore, perhaps generators can be handled by a strictly local rewriting process. That is, in the core language generators can be reified into first-class functions and applications that use a little state to record which function is the continuation of the `yield` point. Thus, generators seem to fit perfectly with our desugaring strategy.

To convert programs to CPS, we take operators that can cause control-flow and reify each into a continuation function and appropriate application. These operators include simple sequences, loops combined with `break` and `continue`, `try-except` and `try-finally` combined with `raise`, and `return`. Our CPS transformation turns every expression into a function that accepts an argument for each of the above control operators, and turns uses of control operators into applications of the appropriate continuation inside the function. By passing in different continuation arguments, the caller of the resulting function has complete control over the behavior of control operators. For example, we might rewrite a `try-except` block from

```
try:
  raise Exception()
except e:
  print(e)
```

to

```
def except_handler(e): print(e)
except_handler(Exception())
```

In the case of generators, rewriting `yield` with CPS would involve creating a handler that stores a function holding the code for what to do next, and rewriting `yield` expressions to call that handler:

```
def f():
  x = 1
  yield x
  x += 1
  yield x

g = f()
g.__next__() # ==> 1
g.__next__() # ==> 2
g.__next__() # throws "StopIteration"
```

would be rewritten to something like:[6]

```
def f():

  def yielder(val, rest_of_function):
    next.to_call_next = rest_of_function
    return val

  def next():
    return next.to_call_next()

  def done(): raise StopIteration()
  def start():
    x = 1
    def rest():
      x += 1
      return yielder(x, done)
    return yielder(x, rest)

  next.to_call_next = start

  return { 'next' : next }
```

[6] This being a sketch, we have taken some liberties for simplicity.

```
g = f()
g['next']() # ==> 1
g['next']() # ==> 2
g['next']() # throws "StopIteration"
```

We build the `yielder` function, which takes a value, which it returns after storing a continuation argument in the `to_call_next` field. The `next` function always returns the result of calling this stored value. Each `yield` statement is rewritten to put everything after it into a new function definition, which is passed to the call to `yielder`. In other words, this is the canonical CPS transformation, applied in the usual fashion.

This rewriting is well-intentioned but does not work. If this program is run under Python, it results in an error:

```
    x += 1
UnboundLocalError: local variable 'x'
```

This is because Python creates a *new scope* for each function definition, and assignments within that scope create new variables. In the body of `rest`, the assignment `x += 1` refers to a new `x`, not the one defined by `x = 1` in `start`. This runs counter to traditional notions of functions that can close over mutable variables. And in general, with multiple assignment statements and branching control flow, it is entirely unclear whether a CPS transformation to Python function definitions can work.

The lesson from this example is that the *interaction* of non-traditional scope and control flow made a traditional translation not work. The straightforward CPS solution is thus incorrect in the presence of Python's mechanics of variable binding. We now move on to describing how we can express Python's scope in a more traditional lexical model. Then, in section 4.3 we will demonstrate a working transformation for Python's generators.

4.2 Scope

Python has a rich notion of scope, with several types of variables and implicit binding semantics that depend on the block structure of the program. Most identifiers are `local`; this includes function parameters and variables defined with the = operator. There are also `global` and `nonlocal` variables, with their own special semantics within closures, and interaction with classes. Our core contribution to explaining Python's scope is to give a desugaring of the `nonlocal` and `global` keywords, along with implicit `local`, `global` and `instance` identifiers, into traditional lexically scoped closures. Global scope is still handled specially, since it exhibits a form of dynamic scope that isn't straightforward to capture with traditional let-bindings.[7]

[7] We actually exploit this dynamic scope in bootstrapping Python's object system, but defer an explanation to the appendix.

We proceed by describing Python's handling of scope for local variables, the extension to `nonlocal`, and the interaction of both of these features with classes.

4.2.1 Declaring and Updating Local Variables

In Python, the operator = performs local variable binding:

```
def f():
  x = 'local variable'
  return x

f() # ==> 'local variable'
```

The syntax for updating and creating a local variable are identical, so subsequent = statements mutate the variable created by the first.

```
def f():
  x = 'local variable'
  x = 'reassigned'
  x = 'reassigned again'
  return x

f() # ==> 'reassigned again'
```

Crucially, there is *no syntactic difference* between a statement that updates a variable and one that initially binds it. Because bindings can also be introduced inside branches of conditionals, it isn't statically determinable if a variable will be defined at certain program points. Note also that variable declarations are not scoped to all blocks—here they are scoped to function definitions:

```
def f(y):
 if y > .5: x = 'big'
 else : pass
 return x

f(0) # throws an exception
f(1) # ==> "big"
```

Handling simple declarations of variables and updates to variables is straightforward to translate into a lexically-scoped language. λ_π has a usual `let` form that allows for lexical binding. In desugaring, we scan the body of the function and accumulate all the variables on the left-hand side of assignment statements in the body. These are let-bound at the top of the function to the special ☠ form, which evaluates to an exception in any context other than a `let`-binding context (section 2). We use `x := e` as the form for variable assignment, which is not a binding form in the core. Thus, in λ_π, the example above rewrites to:

```
let  f = ☠  in
 f :=
 fun (y) (no-var)
  let  x = ☠  in
   if (builtin-prim "num>" (y ⟨%float,0.5⟩))
      x := ⟨%str,"big"⟩
      none
  (return x)

 f (⟨%int,0⟩)
 f (⟨%int,1⟩)
```

In the first application (to 0) the assignment will never happen, and the attempt to look up the ☠-valued *x* in the return statement will fail with an exception. In the second application, the assignment in the then-branch will change the value of *x* in the store to a non-☠ string value, and the string "big" will be returned.

The algorithm for desugaring scope is so far:

- For each function body:
  - Collect all variables on the left-hand side of = in a set *locals*, stopping at other function boundaries,
  - For each variable var in *locals*, wrap the function body in a let-binding of var to ☠.

This strategy works for simple assignments that may or may not occur within a function, and maintains lexical structure for the possibly-bound variables in a given scope. Unfortunately, this covers only the simplest cases of Python's scope.

4.2.2 Closing Over Variables

Bindings from outer scopes can be *seen* by inner scopes:

```
def f():
  x = 'closed-over'
  def g():
    return x
  return g

f()() # ==> 'closed-over'
```

However, since = defines a new local variable, one cannot close over a variable and mutate it with what we've seen so far; = simply defines a new variable with the same name:

```
def g():
  x = 'not affected'
  def h():
    x = 'inner x'
    return x
  return (h(), x)

g() # ==> ('inner x', 'not affected')
```

This is mirrored in our desugaring: each function adds a new let-binding inside its own body, shadowing any bindings from outside. This was the underlying problem with the attempted CPS translation from the last section, highlighting the consequences of using the same syntactic form for both variable binding and update.

Closing over a mutable variable is, however, a common and useful pattern. Perhaps recognizing this, Python added a new keyword in Python 3.0 to allow this pattern, called nonlocal. A function definition can include a nonlocal declaration at the top, which allows mutations within the function's body to refer to variables in enclosing scopes on a per-variable basis. If we add such a declaration to the previous example, we get a different answer:

```
def g():
  x = 'not affected by h'
  def h():
    nonlocal x
    x = 'inner x'
    return x
  return (h(), x)

g() # ==> ('inner x', 'inner x')
```

The nonlocal declaration allows the inner assignment to x to "see" the outer binding from g. This effect can span any nesting depth of functions:

```
def g():
  x = 'not affected by h'
  def h():
    def h2():
      nonlocal x
      x = 'inner x'
      return x
    return h2
  return (h()(), x)

g() # ==> ('inner x', 'inner x')
```

Thus, the presence or absence of a nonlocal declaration can change an assignment statement from a binding occurrence of a variable to an assigning occurrence. We augment our algorithm for desugaring scope to reflect this:

- For each function body:
  - Collect all variables on the left-hand side of = in a set *locals*, stopping at other function boundaries,
  - Let *locals'* be *locals* with any variables in nonlocal declarations removed,
  - For each variable var in *locals'*, wrap the function body in a let-binding of var to ☠.

Thus the above program would desugar to the following, which *does not* let-bind *x* inside the body of the function assigned to *h*.

```
def f(x, y):
  print(x); print(y); print("")
  class c:
    x = 4
    print(x); print(y)
    print("")
    def g(self):
      print(x); print(y); print(c)
  return c

f("x-value", "y-value")().g()

# produces this result:

x-value
y-value

4
y-value

x-value
y-value
<class '__main__.c'>
```

Figure 8: An example of class and function scope interacting

```
let  g = ☠  in
 g :=
 fun ()
  let  x = ☠  in
   let  h = ☠  in
    x := ⟨%str,“not affected by h”,{}⟩
    h :=
    fun ()
     x := ⟨%str,“inner x”,{}⟩
     (return x)
    (return  tuple⟨%tuple,(h () () x)⟩)
 g ()
```

The assignment to *x* inside the body of *h* behaves as a typical assignment statement in a closure-based language like Scheme, mutating the let-bound *x* defined in *g*.

4.2.3 Classes and Scope

We've covered some subtleties of scope for local and nonlocal variables and their interaction with closures. What we've presented so far would be enough to recover lexical scope for a CPS transformation for generators if function bodies contained only other functions. However, it turns out that we observe a different closure behavior for variables in a class definition than we do for variables elsewhere in Python, and we must address classes to wrap up the story on scope.

Consider the example in figure 8. Here we observe an interesting phenomenon: in the body of `g`, the value of the variable `x` is *not* `4`, the value of the most recent apparent assignment to `x`. In fact, the body of `g` seems to "skip" the scope in which x is bound to 4, instead closing over the outer scope in which `x` is bound to `"x-value"`. At first glance this does not appear to be compatible with our previous notions of Python's closures. We will see, however, that the correct desugaring is capable of expressing the semantics of scope in classes within the framework we have already established for dealing with Python's scope.

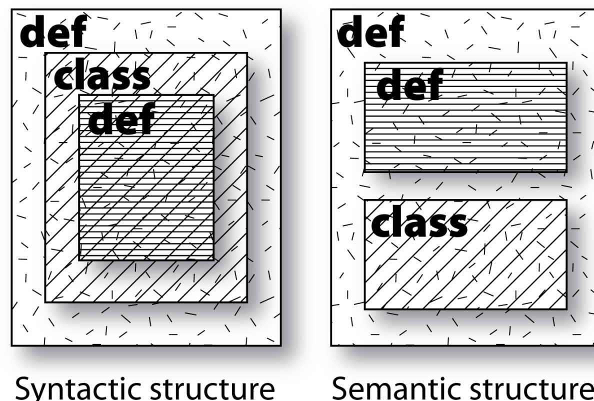

Figure 9: Interactions between class bodies and function scope

Desugaring classes is substantially more complicated than handling simple local and nonlocal cases. Consider the example from figure 8, stripped of print statements:

```
def f(x, y):
  class c:
    x = 4
    def g(self): pass
  return c
f("x-value", "y-value")().g()
```

In this example, we have three local scopes: the body of the function f, the body of the class definition c, and the body of the function g. The scopes of `c` and `g` close over the same scope as `f`, but have distinct, non-nesting scopes themselves. Figure 9 shows the relationship graphically. Algorithmically, we add new steps to scope desugaring:

- For each function body:
  - For each nested class body:
    - Collect all the function definitions within the class body. Let the set of their names be `defnames` and the set of the expressions be `defbodies`,
    - Generate a fresh variable `deflifted` for each variable in `defnames`. Add assignment statements to the function body just before the class definition as-

signing each deflifted to the corresponding expression in defbodies,

  – Change each function definition in the body of the class to defname := deflifted

- Collect all variables on the left-hand side of = in a set *locals*, stopping at other function boundaries,
- Let *locals'* be *locals* with any variables in nonlocal declarations removed,
- For each variable var in *locals'*, wrap the function body in a let-binding of var to ☠.

Recalling that the instance variables of a class desugar roughly to assignments to the class object itself, the function desugars to the following:

```
let  f = ☠  in
 f :=
 fun (x y)
   let  extracted-g = ☠  in
   let  c = ☠  in
      extracted-g :=  fun () none
      c := (class "c")
      c[⟨%str,"x"⟩ := ⟨%int,4⟩]
      c[⟨%str,"g"⟩ := extracted-g]
      (return c)
```

This achieves our desired semantics: the bodies of functions defined in the class C will close over the x and y from the function definition, and the statements written in c-scope can still see those bindings. We note that scope desugaring yields terms in an intermediate language with a class keyword. In a later desugaring step, we remove the class keyword as we describe in section 3.2.

4.2.4 Instance Variables

When we introduced classes we saw that there is no apparent difference between classes that introduce identifiers in their body and classes that introduce identifiers by field assignment. That is, either of the following forms will produce the same class C:

```
class C:
  x = 3
# or ...
class C: pass
C.x = 3
```

We do, however, have to still account for *uses* of the instance variables inside the class body, which are referred to with the variable name, not with a field lookup like c.x. We perform a final desugaring step for instance variables, where we let-bind them in a new scope just for evaluating the class body, and desugar each instance variable assignment into both a class assignment and an assignment to the variable itself. The full desugaring of the example is shown in figure 10.

```
let  f = ☠  in
 f :=
 fun (x y)
   let  extracted-g = ☠  in
   let  c = ☠  in
      extracted-g :=  fun () none
      c := (class "c")
       let  g = ☠  in
       let  x = ☠  in
          c[⟨%str,"x"⟩ := ⟨%int,4⟩]
          x := ⟨%int,4⟩
          c[⟨%str,"g"⟩ := extracted-g]
          g := extracted-g
      (return c)
```

Figure 10: Full class scope desugaring

We have now covered Python classes' scope semantics: function bodies do not close over the class body's scope, class bodies create their own local scope, statements in class bodies are executed sequentially, and definitions/assignments in class bodies result in the creation of class members. The nonlocal keyword does not require further special treatment, even when present in class bodies.

4.3 Generators Redux

With the transformation to a lexical core in hand, we can return to our discussion of generators, and their implementation with a local CPS transformation.

To implement generators, we first desugar Python down to a version of λ_π with an explicit yield statement, passing yields through unchanged. As the final stage of desugaring, we identify functions that contain yield, and convert them to generators via local CPS. We show the desugaring machinery *around* the CPS transformation in figure 11. To desugar them, in the body of the function we construct a generator object and store the CPS-ed body as a ___resume attribute on the object. The __next__ method on the generator, when called, will call the ___resume closure with any arguments that are passed in. To handle yielding, we desugar the core yield expression to update the ___resume attribute to store the current normal continuation, and then return the value that was yielded.

Matching Python's operators for control flow, we have five continuations, one for the normal completion of a statement or expression going onto the next, one for a return statement, one each for break and continue, and one for the exception throwing raise. This means that each CPSed expression becomes an anonymous function of five arguments, and can be passed in the appropriate behavior for each control operator.

We use this configurability to handle two special cases:

- Throwing an exception while running the generator
- Running the generator to completion

```
def f(x ...): body-with-yield
```

desugars to...

```
fun (x ...)
  let done =
    fun () (raise StopIteration ())   in
  let initializer =
    fun (self)
      let end-of-gen-normal =
        fun (last-val)
         self[⟨%str,"__next__"⟩ := done]
         (raise StopIteration ())    in
      let end-of-gen-exn =
        fun (exn-val)
         self[⟨%str,"__next__"⟩ := done]
         (raise exn-val)    in
      let unexpected-case =
        fun (v)
         (raise SyntaxError ())   in
      let resumer =
        fun (yield-send-value)
         (return
           (cps-of body-with-yield)
           (end-of-gen-normal
            unexpected-case
            end-of-gen-exn
            unexpected-case
            unexpected-case))   in
     self[⟨%str,"___resume"⟩ := resumer]       in
   %generator (initializer)
```

where...

```
class generator(object):
    def __init__(self, init):
        init(self)

    def __next__(self):
        return self.___resume(None)

    def send(self, arg):
        return self.___resume(arg)

    def __iter__(self):
        return self

    def __list__(self):
        return [x for x in self]
```

Figure 11: The desugaring of generators

In the latter case, the generator raises a `StopIteration` exception. We encode this by setting the initial "normal" continuation to a function that will update `___resume` to always raise `StopIteration`, and then to raise that exception. Thus, if we evaluate the entire body of the generator, we will pass the result to this continuation, and the proper behavior will occur.

Similarly, if an uncaught exception occurs in a generator, the generator will raise that exception, and any subsequent calls to the generator will result in `StopIteration`. We handle this by setting the initial `raise` continuation to be code that updates `___resume` to always raise `StopIteration`, and then we raise the exception that was passed to the continuation. Since each `try` block in CPS installs a new exception continuation, if a value is passed to the top-level exception handler it means that the exception was not caught, and again the expected behavior will occur.

5. Engineering & Evaluation

Our goal is to have desugaring and λ_π enjoy two properties:

- Desugaring translates all Python source programs to λ_π (*totality*).
- Desugared programs evaluate to the same value as the source would in Python (*conformance*).

The second property, in particular, cannot be proven because there is no formal specification for what Python does. We therefore tackle both properties through testing. We discuss various aspects of implementing and testing below.

5.1 Desugaring Phases

Though we have largely presented desugaring as an atomic activity, the paper has hinted that it proceeds in phases. Indeed, there are four:

- Lift definitions out of classes (section 4.2.3).
- Let-bind variables (section 4.2.2). This is done second to correctly handle occurrences of `nonlocal` and `global` in class methods. The result of these first two steps is an intermediate language between Python and the core with lexical scope, but still many surface constructs.
- Desugar classes, turn Python operators into method calls, turn `for` loops into guarded `while` loops, etc.
- Desugar generators (section 4.3).

These four steps yield a term in our core, but it isn't ready to run yet because we desugar to open terms. For instance, `print(5)` desugars to

```
print (⟨%int,5⟩)
```

which relies on free variables `print` and `%int`.

5.2 Python Libraries in Python

We implement as many libraries as possible in Python[8] augmented with some macros recognized by desugaring. For example, the builtin tuple class is implemented in Python, but getting the length of a tuple defers to the δ function:

```
class tuple(object):
  def __len__(self):
    return ___delta("tuple-len", self)
  ...
```

All occurrences of `___delta(str, e, ...)` are desugared to `(builtin-prim str (e ...))` directly. We only do this for *library* files, so normal Python programs can use `___delta` as the valid identifier it is. As another example, after the class definition of tuples, we have the statement

```
___assign("%tuple", tuple)
```

which desugars to an assignment statement `%tuple := tuple`. Since %-prefixed variables aren't valid in Python, this gives us an private namespace of global variables that are un-tamperable by Python. Thanks to these decisions, this project produces far more readable desugaring output than a previous effort for JavaScript [6].

5.3 Performance

λ_π may be intended as a formal semantics, but composed with desugaring, it also yields an implementation. While the performance does not matter for semantic reasons (other than programs that depend on time or space, which would be ill-suited by this semantics anyway), it does greatly affect how quickly we can iterate through testing!

The full process for running a Python program in our semantics is:

1. Parse and desugar roughly 1 KLOC of libraries implemented in Python
2. Parse and desugar the target program
3. Build a syntax tree of several built-in libraries, coded by building the AST directly in Racket
4. Compose items 1-3 into a single λ_π expression
5. Evaluate the λ_π expression

Parsing and desugaring for (1) takes a nontrivial amount of time (40 seconds on the first author's laptop). Because this work is needlessly repeated for each test, we began caching the results of the desugared library files, which reduced the testing time into the realm of feasibility for rapid development. When we first performed this optimization, it made running 100 tests drop from roughly 7 minutes to 22 seconds. Subsequently, we moved more functionality out of λ_π and into verbose but straightforward desugaring, causing serious performance hit; running 100 tests now takes on the order of 20 minutes, even with the optimization.

[8] We could not initially use existing implementations of these in Python for bootstrapping reasons: they required more of the language than we supported.

Feature	# of tests	LOC
Built-in Datatypes	81	902
Scope	39	455
Exceptions	25	247
(Multiple) Inheritance	16	303
Properties	9	184
Iteration	13	214
Generators	9	129
Modules	6	58
Total	205	2636

Figure 12: Distribution of passing tests

5.4 Testing

Python comes with an extensive test suite. Unfortunately, this suite depends on numerous advanced features, and as such was useless as we were building up the semantics. We therefore went through the test suite files included with CPython, April 2012,[9] and ported a representative suite of 205 tests (2600 LOC). In our selection of tests, we focused on orthogonality and subtle corner-cases. The distribution of those tests across features is reported in figure 12. On all these tests *we obtain the same results as CPython*.

It would be more convincing to eventually handle all of Python's own `unittest` infrastructure to run CPython's test suite unchanged. The `unittest` framework of CPython unfortunately relies on a number of reflective features on modules, and on native libraries, that we don't yet cover. For now, we manually move the assertions to simpler if-based tests, which also run under CPython, to check conformance.

5.5 Correspondence with Redex

We run our tests against $\lambda_{\pi\downarrow}$, not against the Redex-defined reduction relation for λ_π. We can run tests on λ_π, but performance is excruciatingly slow: it takes over an hour to run complete individual tests under the Redex reduction relation. Therefore, we have been able to perform only limited testing for conformance by hand-writing portions of the environment and heap (as Redex terms) that the Python code in the test uses. Fortunately, executing against Redex should be parallelizable, so we hope to increase confidence in the Redex model as well.

[9] http://www.python.org/getit/releases/3.2.3/

6. Future Work and Perspective

As section 5 points out, there are some more parts of Python we must reflect in the semantics before we can run Python's test cases in their native form. This is because Python is a large language with extensive libraries, a foreign-function interface, and more.

Libraries aside, there are some interesting features in Python left to tackle. These include special fields, such as the properties of function objects that compute the content of closures, complex cases of destructuring assignments, a few reflective features like the metaclass form, and others.

More interestingly, we are not done with scope! Consider `locals`, which returns a dictionary of the current variable bindings in a given scope:

```
def f(x):
  y = 3
  return locals()

f("val") # ==> {'x': 'val', 'y': 3}
```

This use of `locals` can be desugared to a clever combination of assignments into a dictionary along with variable assignments, which we do. However, this desugaring of `locals` relies on it being a strictly *local* operation (for lack of a better word). But worse, `locals` is a value!

```
def f(x, g):
  y = 3
  return g()

f("x-val", locals)
# ==> {'x': 'x-val', 'y': 3,
#       'g': <builtin function locals>}
```

Thus, *any* application could invoke `locals`. We would therefore need to deploy our complex desugaring everywhere we cannot statically determine that a function is not `locals`, and change every application to check for it. Other built-in values like `super` and `dir` exhibit similar behavior.

On top of this, `import` can splice all identifiers (`*`) from a module into local scope. For now, we handle only `import`s that bind the module object to a single identifier. Indeed, even Python 3 advises that `import *` should only be used at module scope. Finally, we do not handle `exec`, Python's "eval" (though the code-loading we do for modules comes close). Related efforts on handling similar operators in JavaScript [6] are sure to be helpful here.

We note that most traditional analyses would be seriously challenged by programs that use functions like `locals` in a higher-order way, and would probably benefit from checking that it isn't used in the first place. We don't see the lack of full support for such functions as a serious weakness of λ_π, or an impediment to reasoning about most Python programs. Rather, it's an interesting future challenge to handle a few of these remaining esoteric features. It's also useful to simply call out the weirdness of these operators, which are liable to violate the soundness of otherwise-sound program tools.

Overall, what we have learned most from this effort is how central scope is to understanding Python. Many of its features are orthogonal, but they all run afoul on the shoals of scope. Whether this is intentional or an accident of the contorted history of Python's scope is unclear (for example, see the discussion around the proposal to add `nonlocal` [15]), but also irrelevant. Those attempting to improve Python or create robust sub-languages of it—whether for teaching or for specifying asset-backed securities—would do well to put their emphasis on scope first, because this is the feature most likely to preclude sound analyses, correctness-preserving refactorings, and so on.

7. Related Work

We are aware of only one other formalization for Python: Smeding's unpublished and sadly unheralded master's thesis [9]. Smeding builds an executable semantics and tests it against 134 hand-written tests. The semantics is for Python 2.5, a language version without the complex scope we handle. Also, instead of defining a core, it directly evaluates (a subset of) Python terms. Therefore, it offers a weaker account of the language and is also likely to be less useful for certain kinds of tools and for foundational work.

There are a few projects that analyze Python code. They are either silent about the semantics or explicitly eschew defining one. We therefore do not consider these related.

Our work follows the path laid out by λ_{JS} [7] and its follow-up [6], both for variants of JavaScript.

Acknowledgments

We thank the US NSF and Google for their support. We thank Caitlin Santone for lending her graphical expertise to figure 9. The paper title is entirely due to Benjamin Lerner. We are grateful to the Artifact Evaluation Committee for their excellent and detailed feedback, especially the one reviewer who found a bug in iteration with generators.

This paper was the result of an international collaboration resulting from an on-line course taught by the first and last authors. Several other students in the class also contributed to the project, including Ramakrishnan Muthukrishnan, Bo Wang, Chun-Che Wang, Hung-I Chuang, Kelvin Jackson, Victor Andrade, and Jesse Millikan.

Bibliography

[1] Appcelerator. PyDev. 2013. http://pydev.org/

[2] Johann C. Rocholl. PEP 8 1.4.5. 2013. https://pypi.python.org/pypi/pep8

[3] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. 2009.

[4] Matthias Felleisen and Robert Hieb. The Revised Report on the Syntactic Theories of Sequential Control and State. *Theoretical Computer Science* 103(2), 1992.

[5] Phil Frost. Pyflakes 0.6.1. 2013. `https://pypi.python.org/pypi/pyflakes`

[6] Joe Gibbs Politz, Matthew J. Carroll, Benjamin S. Lerner, Justin Pombrio, and Shriram Krishnamurthi. A Tested Semantics for Getters, Setters, and Eval in JavaScript. In *Proc. Dynamic Languages Symposium*, 2012.

[7] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The Essence of JavaScript. In *Proc. European Conference on Object-Oriented Programming*, 2010.

[8] JetBrains. PyCharm. 2013. `http://www.jetbrains.com/pycharm/`

[9] Gideon Joachim Smeding. An executable operational semantics for Python. Universiteit Utrecht, 2009.

[10] James McCauley. About POX. 2013. `http://www.noxrepo.org/pox/about-pox/`

[11] Neal Norwitz. PyChecker 0.8.12. 2013. `https://pypi.python.org/pypi/PyChecker`

[12] Securities and Exchange Commission. Release Nos. 33-9117; 34-61858; File No. S7-08-10. 2010. `https://www.sec.gov/rules/proposed/2010/33-9117.pdf`

[13] Sylvain Thenault. PyLint 0.27.0. 2013. `https://pypi.python.org/pypi/pylint`

[14] Wingware. Wingware Python IDE. 2013. `http://wingware.com/`

[15] Ka-Ping Yee. Access to Names in Outer Spaces. 2009. `http://www.python.org/dev/peps/pep-3104/`

Appendix 1: The Rest of λ_π

The figures in this section show the rest of the λ_π semantics. We proceed to briefly present each of them in turn.

1. Contexts and Control Flow

Figures 13, 14, 15, and 16 show the different *contexts* we use to capture left-to-right, eager order of operations in λ_π. *E* is the usual *evaluation context* that enforces left-to-right, eager evaluation of expressions. *T* is a context for the first expression of a `tryexcept` block, used to catch instances of `raise`. Similarly, *H* defines contexts for loops, detecting `continue` and `break`, and *R* defines contexts for `return` statements inside functions. Each interacts with a few expression forms to handle non-local control flow.

Figure 17 shows how these contexts interact with expressions. For example, in first few rules, we see how we handle `break` and `continue` in `while` statements. When `while` takes a

```
E ::= []
    | (fetch E) | (set! E e) | (set! val E)
    | (alloc E)
    | (module E e)
    | ⟨E,mval⟩
    | ref := E  | x := E
    | E[e := e] | v[E := e] | v[v := E]
    | E e
    |  if E e e
    |  let  x = E  in e
    |  list⟨E,[e ...]⟩ |  list⟨val,Es⟩
    |  tuple⟨E,(e ...)⟩ |  tuple⟨val,Es⟩
    |  set⟨E,(e ...)⟩ |  set⟨val,Es⟩
    | E[e] | val[E]
    | (builtin-prim op Es)
    | (raise E)
    | (return E)
    | (tryexcept E x e e) | (tryfinally E e)
    | (loop E e) | (frame E)
    | E (e ...) | val Es
    | E (e ...)*e | val Es*e
    | val (val ...)*E
    | (construct-module E) | (in-module E ε)
Es ::= (val ... E e ...)
```

Figure 13: Evaluation contexts

```
T ::= []
    | (fetch T) | (set! T e) | (set! val T)
    | (alloc T)
    | ⟨T,mval⟩
    | ref := T  | x := T
    | T[e := e] | v[T := e] | v[v := T]
    | T e
    |  if T e e
    |  let  x = T  in e
    |  list⟨T,e⟩ |  list⟨val,Ts⟩
    |  tuple⟨T,e⟩ |  tuple⟨val,Ts⟩
    |  set⟨T,e⟩ |  set⟨val,Ts⟩
    | T[e] | val[T]
    | (builtin-prim op Ts)
    | (raise T)
    | (loop T e) | (frame T)
    | T (e ...) | val Ts
    | T (e ...)*e | val Ts*e
    | val (val ...)*T
    | (construct-module T)
Ts ::= (val ... T e ...)
```

Figure 14: Contexts for try-except

```
H ::= []
    | (fetch H) | (set! H e) | (set! val H)
    | (alloc H)
    | ⟨H, mval⟩
    | ref := H  | x := H
    | H[e := e] | v[H := e] | v[v := H]
    | H e
    | if H e e
    | let x = H in e
    | list⟨H,e⟩ | list⟨val,Hs⟩
    | tuple⟨H,e⟩ | tuple⟨val,Hs⟩
    | set⟨H,e⟩ | set⟨val,Hs⟩
    | H[e] | val[H]
    | (builtin-prim op Hs)
    | (raise H)
    | (tryexcept H x e e)
    | H (e ...) | val Hs
    | H (e ...)*e | val Hs*e
    | val (val ...)*H
    | (construct-module E)
Hs ::= (val ... H e ...)
```

Figure 15: Contexts for loops

```
R ::= []
    | (fetch R) | (set! R e) | (set! val R)
    | (alloc R)
    | ⟨R, mval⟩
    | ref := R  | x := R
    | R[e := e] | v[R := e] | v[v := R]
    | R e
    | if R e e
    | let x = R in e
    | list⟨R,e⟩ | list⟨val,Rs⟩
    | tuple⟨R,e⟩ | tuple⟨val,Rs⟩
    | set⟨R,e⟩ | set⟨val,Rs⟩
    | R[e] | val[R]
    | (builtin-prim op Rs)
    | (raise R)
    | (loop R e)
    | (tryexcept R x e e)
    | R (e ...) | val Rs
    | R (e ...)*e | val Rs*e
    | val (val ...)*R
    | (construct-module E)
Rs ::= (val ... R e ...)
```

Figure 16: Contexts for return statements

```
(while e1 e2 e3)                                          [E-While]
⟹ if e1 (loop e2 (while e1 e2 e3)) e3

(loop H[continue] e) ⟹ e                                  [E-LoopContinue]

(loop H[break] e) ⟹ vnone                                 [E-LoopBreak]

(loop val e) ⟹ e                                          [E-LoopNext]

(tryexcept val x e_catch e_else) ⟹ e_else                 [E-TryDone]

(tryexcept T[(raise val)] x e_catch e_else)               [E-TryCatch]
⟹ let x = val in e_catch

(frame R[(return val)]) ⟹ val                             [E-Return]

(frame val) ⟹ val                                         [E-FramePop]

(tryfinally R[(return val)] e)                            [E-FinallyReturn]
⟹ e (return val)

(tryfinally H[break] e)                                   [E-FinallyBreak]
⟹ e break

(tryfinally T[(raise val)] e)                             [E-FinallyRaise]
⟹ e (raise val)

(tryfinally H[continue] e)                                [E-FinallyContinue]
⟹ e continue

(E[ if val e1 e2 ] ε Σ)                                   [E-IfTrue]
⟶ (E[e1] ε Σ)
  where (truthy? val Σ)

(E[ if val e1 e2 ] ε Σ)                                   [E-IfFalse]
⟶ (E[e2] ε Σ)
  where (not (truthy? val Σ))

val e ⟹ e                                                 [E-Seq]
```

Figure 17: Control flow reductions

step, it yields a `loop` form that serves as the marker for where internal `break` and `continue` statements should collapse to. It is for this reason that H does *not* descend into nested `loop` forms; it would be incorrect for a `break` in a nested loop to `break` the outer loop.

One interesting feature of `while` and `tryexcept` in Python is that they have distinguished "else" clauses. For `while` loops, these else clauses run when the condition is `False`, but *not* when the loop is broken out of. For `tryexcept`, the else clause is only visited if *no* exception was thrown while evaluating the body. This is reflected in E-TryDone and the else branch of the `if` statement produced by E-While.

We handle one feature of Python's exception raising imperfectly. If a programmer uses `raise` without providing an explicit value to throw, *the exception bound in the most recent active catch block* is thrown instead. We have a limited solution that involves raising a special designated "reraise" value, but this fails to capture some subtle behavior of nested catch blocks. We believe a more sophisticated desugaring that uses a global stack to keep track of entrances and exits to catch blocks will work, but have yet to verify it. We still pass a number of tests that use `raise` with no argument.

2. Mutation

There are *three* separate mutability operators in λ_π, `(set! e e)`, which mutates the value stored in a reference value, `e := e`, which mutates variables, and `(set-field e e e)`, which updates and adds fields to objects.

Figure 18 shows the several operators that allocate and manipulate references in different ways. We briefly categorize the purpose for each type of mutation here:

- We use `(set! e e)`, `(fetch e)` and `(alloc e)` to handle the update and creation of objects via the δ function, which reads but does not modify the store. Thus, even the lowly `+` operation needs to have its result re-allocated, since programmers only see references to numbers, not object values themselves. We leave the pieces of object values immutable and use this general strategy for updating them, rather than defining separate mutability for each type (e.g., lists).
- We use `e := e` for assignment to both local and global variables. We discuss global variables more in the next section. Local variables are handled at binding time by allocating references and substituting the new references wherever the variable appears. Local variable accesses and assignments thus work over references directly, since the variables have been substituted away by the time the actual assignment or access is reached. Note also that E-AssignLocal can override potential ☠ store entries.
- We use `(set-field e e e)` to update and add fields to objects' dictionaries. We leave the fields of objects' dictionaries as references and not values to allow ourselves the ability to share references between object fields and variables. We maintain a strict separation in our current semantics, with the exception of modules, and we expect that we'll continue to need it in the future to handle patterns of `exec`.

Finally, we show the E-Delete operators, which allow a Python program to revert the value in the store at a particular location back to ☠, or to remove a global binding.

3. Global Scope

While local variables are handled directly via substitution, we handle global scope with an explicit environment ε that follows the computation. We do this for two main reasons. First, because global scope in Python is truly dynamic in ways that local scope is not (`exec` can modify global scope), and we want to be open to those possibilities in the future. Second, and more implementation-specific, we use global scope to bootstrap some mutual dependencies in the object system, and allow ourselves a touch of dynamism in the semantics.

For example, when computing booleans, λ_π needs to yield numbers from the δ function that are real booleans (e.g., have the built-in `%bool` object as their class). However, we need booleans to set up if-tests while we are bootstrapping the creation of the boolean class itself! To handle this, we allow global identifiers to appear in the class position of objects. If we look for the class of an object, and the class position is an identifier, we look it up in the global environment. We only use identifiers with special %-prefixed names that aren't visible to Python in this way. It may be possible to remove this touch of dynamic scope from our semantics, but we haven't yet found the desugaring strategy that lets us do so. Figure 19 shows the reduction rule for field lookup in this case.

4. `True`, `False`, and `None`

The keywords `True`, `False`, and `None` are all singleton references in Python. In λ_π, we do not have a form for `True`, instead desugaring it to a variable reference bound in the environment. The same goes for `None` and `False`. We bind each to an allocation of an object:

```
let  True = (alloc ⟨%bool,1,{}⟩)  in
let  False = (alloc ⟨%bool,0,{}⟩)  in
let  None = (alloc ⟨%none, meta-none ,{}⟩)  in
 e_prog
```

and these bindings happen before anything else. This pattern ensures that all references to these identifiers in the desugared program are truly to the same objects. Note also that the boolean values are represented simply as number-like values, but with the built-in `%bool` class, so they can be added and subtracted like numbers, but perform method lookup on the `%bool` class. This reflects Python's semantics:

```
isinstance(True, int) # ==> True
```

$$(E[\texttt{@}ref_{fun}\ \ (val\ \texttt{...})]\ \varepsilon\ \Sigma) \qquad \text{[E-App]}$$

$$\longrightarrow (E[(\texttt{frame}\ \text{subst}[\![(x\ \texttt{...}),\ (ref_{arg}\ \texttt{...}),\ e]\!])]\ \varepsilon\ \Sigma_1)$$

$$\texttt{where}\ \langle any_c, \lambda(x\ \texttt{...})\ (\texttt{no-var}).e\ \ , any_{dict}\rangle = \Sigma(ref_{fun}),$$
$$(\texttt{equal?}\ (\texttt{length}\ (val\ \texttt{...}))\ \ (\texttt{length}\ (x\ \texttt{...}))),$$
$$(\Sigma_1\ \ (ref_{arg}\ \texttt{...}_1)) = \text{extend-store/list}[\![\Sigma,\ (val\ \texttt{...})]\!]$$

$$(E[\ \texttt{let}\ \ x = v\text{+}undef\ \ \texttt{in}\ e\ ]\ \varepsilon\ \Sigma) \qquad \text{[E-LetLocal]}$$

$$\longrightarrow (E[[x/ref]e\ ]\ \varepsilon\ \Sigma_1)$$

$$\texttt{where}\ (\Sigma_1\ \ ref) = \texttt{alloc}(\Sigma, v\text{+}undef)$$

$$(E[\ \texttt{let}\ \ x = v\text{+}undef\ \ \texttt{in}\ e\ ]\ \varepsilon\ \Sigma) \qquad \text{[E-LetGlobal]}$$

$$\longrightarrow (E[e]\ \text{extend-env}[\![\varepsilon,\ x,\ ref]\!]\ \Sigma_1)$$

$$\texttt{where}\ (\Sigma_1\ \ ref) = \texttt{alloc}(\Sigma, v\text{+}undef)$$

$$(E[ref]\ \varepsilon\ \Sigma) \longrightarrow (E[val]\ \varepsilon\ \Sigma) \qquad \text{[E-GetVar]}$$

$$\texttt{where}\ \Sigma = ((ref_1\ \ v\text{+}undef_1)\ \texttt{...}\ (ref\ \ val)\ (ref_n\ \ v\text{+}undef_n)\ \texttt{...})$$

$$(E[ref]\ \varepsilon\ \Sigma) \longrightarrow (E[val]\ \varepsilon\ \Sigma) \qquad \text{[E-GetVar]}$$

$$\texttt{where}\ \Sigma = ((ref_1\ \ v\text{+}undef_1)\ \texttt{...}\ (ref\ \ val)\ (ref_n\ \ v\text{+}undef_n)\ \texttt{...})$$

$$(E[ref\ \texttt{:=}\ val\ ]\ \varepsilon\ \Sigma) \longrightarrow (E[val]\ \varepsilon\ \Sigma[ref\texttt{:=}val]) \qquad \text{[E-AssignLocal]}$$

$$(E[x\ \texttt{:=}\ val\ ]\ \varepsilon\ \Sigma) \longrightarrow (E[val]\ \varepsilon\ \Sigma[ref\texttt{:=}val]) \qquad \text{[E-AssignGlobal]}$$

$$\texttt{where}\ \varepsilon = ((x_2\ \ ref_2)\ \texttt{...}\ (x\ \ ref)\ (x_3\ \ ref_3)\ \texttt{...})$$

$$(E[(\texttt{alloc}\ val)]\ \varepsilon\ \Sigma) \longrightarrow (E[\texttt{@}ref_{new}\ ]\ \varepsilon\ \Sigma_1) \qquad \text{[E-Alloc]}$$

$$\texttt{where}\ (\Sigma_1\ \ ref_{new}) = \texttt{alloc}(\Sigma, val)$$

$$(E[(\texttt{fetch}\ \texttt{@}ref\ )]\ \varepsilon\ \Sigma) \longrightarrow (E[\Sigma(ref)]\ \varepsilon\ \Sigma) \qquad \text{[E-Fetch]}$$

$$(E[(\texttt{set!}\ \texttt{@}ref\ \ val)]\ \varepsilon\ \Sigma) \longrightarrow (E[val]\ \varepsilon\ \Sigma_1) \qquad \text{[E-Set!]}$$

$$\texttt{where}\ \Sigma_1 = \Sigma[ref\texttt{:=}val]$$

$$(E[\texttt{@}ref_{obj}\ [\texttt{@}ref_{str}\ \ \texttt{:=}\ val_1]]\ \varepsilon\ \Sigma) \longrightarrow (E[val_1]\ \varepsilon\ \Sigma_2) \qquad \text{[E-SetFieldAdd]}$$

$$\texttt{where}\ \langle any_{cls1}, mval, \{string{:}ref, \texttt{...}\}\rangle = \Sigma(ref_{obj}),$$
$$(\Sigma_1\ \ ref_{new}) = \texttt{alloc}(\Sigma, val_1),$$
$$\langle any_{cls2}, string_1, any_{dict}\rangle = \Sigma(ref_{str}),$$
$$\Sigma_2 = \Sigma_1[ref_{obj}\texttt{:=}\langle any_{cls1}, mval, \{string_1{:}ref_{new}, string{:}ref, \texttt{...}\}\rangle],$$
$$(\texttt{not}\ (\texttt{member}\ string_1\ \ (string\ \texttt{...})))$$

$$(E[\texttt{@}ref_{obj}\ [\texttt{@}ref_{str}\ \ \texttt{:=}\ val_1]]\ \varepsilon\ \Sigma) \qquad \text{[E-SetFieldUpdate]}$$

$$\longrightarrow (E[val_1]\ \varepsilon\ \Sigma[ref_1\texttt{:=}val_1])$$

$$\texttt{where}\ \langle any_{cls1}, mval, \{string_2{:}ref_2, \texttt{...}, string_1{:}ref_1, string_3{:}ref_3, \texttt{...}\}\rangle = \Sigma(ref_{obj}),$$
$$\langle any_{cls2}, string_1, any_{dict}\rangle = \Sigma(ref_{str})$$

$$(E[(\texttt{delete}\ ref)]\ \varepsilon\ ((ref_1\ \ v_1)\ \texttt{...}\ (ref\ \ v)\ (ref_n\ \ v_n)\ \texttt{...})) \qquad \text{[E-DeleteLocal]}$$

$$\longrightarrow (E[v]\ \varepsilon\ ((ref_1\ \ v_1)\ \texttt{...}\ (ref\ ☠)\ (ref_n\ \ v_n)\ \texttt{...}))$$

$$(E[(\texttt{delete}\ x)]\ ((x_1\ \ ref_1)\ \texttt{...}\ (x\ \ ref)\ (x_n\ \ ref_n)\ \texttt{...})\ \Sigma) \qquad \text{[E-DeleteGlobal]}$$

$$\longrightarrow (E[\Sigma(ref)]\ ((x_1\ \ ref_1)\ \texttt{...}\ (x_n\ \ ref_n)\ \texttt{...})\ \Sigma)$$

Figure 18: Various operations on mutable variables and values

```
(E[@ref_obj [@ref_str ]] ε Σ)                                           [E-GetField-Class/Id]
⟶ (E[val_result] ε Σ_result)
  where ⟨any_cls,string,any_dict⟩ = Σ(ref_str),
        ⟨x_cls,mval,{string_1:ref_2,...}⟩ = Σ(ref_obj),
        (Σ_result val_result) = class-lookup⟦@ref_obj , Σ(env-lookup⟦ε, x_cls⟧), string, Σ⟧,
        (not (member string (string_1 ...)))
          env-lookup⟦((x_1 ref_1) ... (x ref) (x_n ref_n) ...), x⟧ = ref
```

Figure 19: Accessing fields on a class defined by an identifier

```
(E[@ref_fun (val ...)] ε Σ)                                             [E-AppArity]
⟶ (E[(err ⟨%str,"arity-mismatch",{}⟩)] ε Σ)
  where ⟨any_c,λ(x ...) (no-var).e ,any_dict⟩ = Σ(ref_fun),
        (not (equal? (length (val ...)) (length (x ...))))

(E[@ref_fun (val ...)] ε Σ)                                             [E-AppVarArgsArity]
⟶ (E[(err ⟨%str,"arity-mismatch-vargs",{}⟩)] ε Σ)
  where ⟨any_c,λ(x ...) (y).e ,any_dict⟩ = Σ(ref_fun),
        (< (length (val ...)) (length (x ...)))

(E[@ref_fun (val ...)] ε Σ)                                             [E-AppVarArgs1]
⟶ (E[(frame subst⟦(x ... y_varg), (ref_arg ... ref_tupleptr), e⟧)] ε Σ_3)
  where ⟨any_c,λ(x ...) (y_varg).e ,any_dict⟩ = Σ(ref_fun),
        (>= (length (val ...)) (length (x ...))),
        (val_arg ...) = (take (val ...) (length (x ...))),
        (val_rest ...) = (drop (val ...) (length (x ...))),
        val_tuple = ⟨%tuple,(val_rest ...),{}⟩,
        (Σ_1 (ref_arg ...)) = extend-store/list⟦Σ, (val_arg ...)⟧,
        (Σ_2 ref_tuple) = alloc(Σ_1,val_tuple),
        (Σ_3 ref_tupleptr) = alloc(Σ_2,@ref_tuple )

(E[@ref_fun (val ...)*@ref_var ] ε Σ)                                   [E-AppVarArgs2]
⟶ (E[@ref_fun (val ... val_extra ...)] ε Σ)
  where ⟨any_c,(val_extra ...),any_dict⟩ = Σ(ref_var)
```

Figure 20: Variable-arity functions

5. Variable-arity Functions

We implement Python's variable-arity functions directly in our core semantics, with the reduction rules shown in figure 20. We show first the two arity-mismatch cases in the semantics, where either no vararg is supplied and the argument count is wrong, or where a vararg is supplied but the count is too low. If the count is higher than the number of parameters and a vararg is present, a new tuple is allocated with the extra arguments, and passed as the vararg. Finally, the form `e (e ...)*e` allows a variable-length collection of arguments to be *passed* to the function; this mimics `apply` in languages like Racket or JavaScript.

6. Modules

We model modules with the two rules in figure 21. E-ConstructModule starts the evaluation of the module, which is represented by a `meta-code` structure. A `meta-code` contains a list of global variables to bind for the module, a name for the module, and an expression that holds the module's body. To start evaluating the module, a new location is allocated for each global variable used in the module, initialized to ☠ in the store, and a new environment is created mapping each of these new identifiers to the new locations.

Evaluation that proceeds inside the module, *replacing* the global environment ϵ with the newly-created environment. The old environment is stored with a new `in-module` form that is left in the current context. This step also sets up an

```
   (E[(construct-module @ref_mod )] ε_old Σ)                       [E-ConstructModule]
⟶ (E[(in-module e_body ε_old)
     (alloc ⟨$module,(no-meta),{string_arg:ref_new,...}⟩) ]
     ε_new Σ_1)
    where ⟨any_cls,(meta-code (x_arg ...) x_name e_body),any_dict⟩ = Σ(ref_mod),
          (Σ_1 ε_new (ref_new ...)) = vars->fresh-env⟦Σ, (x_arg ...)⟧,
          (string_arg ...) = (map symbol->string (x_arg ...))

   (E[(in-module v ε_old)] ε_mod Σ) ⟶ (E[vnone] ε_old Σ)        [E-ModuleDone]
vars->fresh-env⟦Σ, ()⟧          = (Σ () ())
vars->fresh-env⟦Σ, (x)⟧         = (Σ_1 ((x ref)) (ref))
 where (Σ_1 (ref)) = (alloc Σ ☠)
vars->fresh-env⟦Σ, (x x_r ...)⟧ = (Σ_2 ((x ref) (x_rest ref_rest) ...) (ref ref_rest ...))
 where (Σ_1 ref) = alloc(Σ,☠),
       (Σ_2 ((x_rest ref_rest) ...) (ref_rest ...)) = vars->fresh-env⟦Σ_1, (x_r ...)⟧
```

Figure 21: Simple modules in λ_π

expression that will create a new module object, whose fields hold the global variable references of the running module.

When the evaluation of the module is complete (the `in-module` term sees a value), the old global environment is reinstated.

To desugar to these core forms, we desugar the files to be imported, and analyze their body to find all the global variables they use. The desugared expression and variables are combined with the filename to create the `meta-code` object. This is roughly an implementation of Python's `compile`, and it should be straightforward to extend it to implement `exec`, though for now we've focused on specifically the module case.

Appendix 2: Confusing Rename Refactorings

This program:

```
def f(x):
  class C(object):
    x = "C's x"
    def meth(self):
      return x + ', ' + C.x
  return C

f('input x')().meth()
# ==> 'input x, C's x'
```

confuses the variable rename refactoring of all the Python IDEs we tried. We present these weaknesses to show that getting a scope analysis right in Python is quite hard! We found these tools by following recommendations on StackOverflow, a trusted resource. Two of the tools we tested, PyCharm and WingWare IDE, are products that developers actually purchase to do Python development (we performed this experiment in their free trials).

For PyCharm, if we rename the `x` parameter to `y`, the class variable `x` also gets changed to `y`, but the access at `C.x` does not. This changes the program to throw an error. If we instead select the `x` in `C.x` and refactor to `y`, The class variable and use of `x` change, but the original definition's parameter does not. This changes the behavior to an error again, as `y` is an unbound identifier in the body of `meth`.

PyDev has the same problem as PyCharm with renaming the function's parameter. If we instead try rename the `x` in the body of `C`, it gets it mostly right, but also renames all the instances of `x` in our strings (something that even a parser, much less a lexical analyzer should be able to detect):

```
def f(x):
  class C(object):
    y = "C's y"
    # we highlighed the x before = above
    # and renamed to y
    def meth(self):
      return x + ' ' + C.y
  return C

f('input y')().meth()
# ==> 'input y, C's y'
```

WingWare IDE for Python is less obviously wrong: it pops up a dialog with different bindings and asks the user to check the ones they want rewritten. However, if we try to refactor based on the `x` inside the method, it doesn't give us an option to rename the function parameter, only the class variable name and the access at `C.x`. In other cases it provides a list that contains a superset of the actual identifiers that should be renamed. In other words, it not only overapproximates (which in Python may be inevitable), it also (more dangerously) undershoots.