CSci 658-01: Software Language Engineering
Python 3 Reflexive Metaprogramming
Chapter 2

H. Conrad Cunningham

4 May 2018

Contents

Professor of Computer and Information Science
University of Mississippi
211 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-5358

Advisory: The HTML version of this document requires use of a browser that supports the display of MathML. A good choice as of May 2018 is a recent version of Firefox from Mozilla.

2 Basic Features Supporting Metaprogramming

In this chapter we examine the characteristics and basic features of Python 3 upon which the reflexive metaprogramming features build.

Python 3 is an *object-oriented* language built on top of an *object-based*, *imperative* language. The language is typically *compiled* to a sequence of instructions for a *virtual machine*, which is then *interpreted*.

TODO: Better introduce the range of content of this chapter

2.1 Objects

All Python 3 data are treated as *objects*.

A Python 3 object has the following *essential* characteristics of objects (as described in the Programming Paradigms notes [Cunningham 2018a]):

a. a *state* (value) drawn from a set of possible values

   The state may consist of several distinct data attributes. In this case, the set of possible values is the Cartesian product of the sets of possible values of each attribute.

b. a set of *operations* that access and/or mutate the state

c. a unique *identity* (e.g., address in memory)

A Python 3 object has one of the two *important but nonessential* characteristics of objects (as described in the Programming Paradigms notes [Cunningham 2018a]). Python 3 does:

d. *not* enforce *encapsulation* of the state within the object, instead relying mostly upon programming conventions to hide private information

e. exhibit an *independent lifecycle* (i.e., has a different lifetime than the code that created it)

As we see in a later section, each object has a distinct dictionary, the directory, that maps the local names to the data attributes and operations.

Python 3 typically uses dot notation to access an object's data attributes and operations.

```
obj.data   # access data attribute of obj
obj.op     # access operation of obj
obj.op()   # invoke operation of obj
```

Some objects are immutable and others are mutable. The *state* (i.e., value) of *immutable* objects (e.g., numbers, booleans, strings, and tuples) cannot be

changed after creation. The state of *mutable* objects (e.g., lists, dictionaries, and sets) can be changed in place after creation.

Caveat: We cannot modify a Python 3 tuple's structure after its creation. However, if the components of a tuple are themselves mutable objects, they can be changed in-place.

All Python 3 objects have a type. What does type mean?

2.2 Type System Concepts

The term "type" tends to be used in many different ways in programming languages.

2.2.1 Types and subtypes

Conceptually, a *type* is a set of values (i.e., possible states) and a set of operations defined on the values in that set.

Similarly, a type `S` is (a behavioral) *subtype* of type `T` if the set of values of type `S` is a "subset" of the values in set `T` and set of operations of type `S` is a "superset" of the operations of type `T`. That is, we can safely *substitute* elements of subtype `S` for elements of type `T` because `S`'s operations behave the "same" as `T`'s operations . This is known as the *Liskov Substitution Principle* [Liskov 1987] [Wikipedia 2018d].

Consider a type representing all furniture and a type representing all chairs. In general, we consider the set of chairs to be a subset of the set of furniture. A chair should have all the general characteristics of furniture, but it may have additional characteristics specific to chairs.

If we can perform an operation on furniture in general, we should be able to perform the same operation on a chair under the same circumstances and get the same result. Of course, there may be additional operations we can perform on chairs that are not applicable to furniture in general.

Thus the type of all chairs is a subtype of the type of all furniture according to the Liskov Substitution Principle.

2.2.2 Constants, variables, and expressions

Now consider the types of the basic program elements.

A *constant* has whatever types it is defined to have in the context in which it is used. For example, the constant symbol `1` might represent an integer, a real number, a complex number, a single bit, etc., depending upon the context.

A *variable* has whatever types its value has at a particular point in time.

An *expression* has whatever types its evaluation yields based on the types of the variables, constants, and operations from which it is constructed.

2.2.3 Static and dynamic

In a *statically typed* language, the types of a variable or expression can be determined from the program source code and checked at "compile time" (i.e., during the syntactic and semantic processing in the front-end of a language processor). Such languages may require at least some of the types of variables or expressions to be *declared* explicitly, while others may be *inferred* implicitly from the context.

Java, Scala, and Haskell are examples of statically typed languages.

In a *dynamically typed language*, the specific types of a variable or expression cannot be determined at "compile time" but can be checked at runtime.

Lisp, Python, and Lua are examples of dynamically typed languages.

Of course, most languages use a mixture of static and dynamic typing. For example, Java objects defined within an inheritance hierarchy must be bound dynamically to the appropriate operations at runtime. Also Java objects declared of type `Object` (the root class of all user-defined classes) often require explicit runtime checks or coercions.

2.2.4 Nominal and structural

In a language with *nominal typing*, the type of an object is based on the type *name* assigned when the object is created. Two objects have the same type if they have the same type name. A type `S` is a subtype of type `T` only if `S` is explicitly declared to be a subtype of `T`.

For example, Java is primarily a nominally typed language. It assigns types to an object based on the name of the class from which the object is instantiated and the superclasses extended and interfaces implemented by that class.

However, Java does not guarantee that subtypes satisfy the Liskov Substitution Principle. For example, a subclass might not implement an operation in a manner that is compatible with the superclass. (The behavior of subclass objects are this different from the behavior of superclass objects.) Ensuring that Java subclasses preserve the Substitution Principle is considered good programming practice in most circumstances.

In a language with *structural typing*, the type of an object based on the *structure* of the object. Two objects have the same type if they have the "same" structure;

that is, they have the same *public* data attributes and operations and these are themselves of compatible types.

In structurally typed languages, a type `S` is a subtype of type `T` only if `S` has all the public data values and operations of type `T` and the data values and operations are themselves of compatible types. Subtype `S` may have additional data values and operations not in `T`.

Haskell is primarily a structurally typed language.

2.3 Python Type System

What about Python 3's type system?

In terms of the discussion in the previous sections, all Python 3 objects can be considered as having one or more conceptual types at a particular point in time. The types may change over time because the program can change the possible set of data attributes and operations associated with the object.

A Python 3 variable is bound to an object by an assignment statement or its equivalent. Python 3 variables are thus dynamically typed, as are Python expressions.

Although a Python 3 program usually constructs an object within a particular nominal type hierarchy (e.g., as an instance of a class), this may not fully describe the type of the object, even initially. And the ability to dynamically add, remove, and modify attributes (both data fields and operations) means the type can change as the program executes.

The type of a Python 3 object is determined by *what it can do* – what data it can hold and what operations it can perform on that data – rather than *how it was created*. We sometimes call this *dynamic, structural typing* approach *duck typing*. (If it walks like a duck and quacks like a duck, then it is a duck, even if is declared as a snake.)

For example, we can say that any object is of an *iterable* type if it implements an `__iter__` operation that returns a valid iterator object. An iterator object must implement a `__next__` operation that retrieves the next element of the "collection" and must raise a `StopIteration` exception if no more elements are available.

In Python 3, we sometimes refer to a type like iterable as a *protocol*. That is, it is a, perhaps informal, *interface* that objects are expected to satisfy in certain circumstances.

2.4 Builtin Types

Python 3 provides several built-in types and subtypes, which are named and implemented in the core language. When displayed, these types are shown as follows:

```
<class 'int'>
```

That is, the value is an instance of a *class* named `int`. Python 3 uses the term *class* to describe its nominal types.

We can query the nominal type of an object `obj` with the function call `type(obj)`. In the following discussion, we show the results from calling this function interactively in Python 3 REPL (Read-Evaluate-Print Loop) sessions.

For the purpose of our discussion, the primary built-in types include:‘

- Singleton types
- Number types
- Sequence types
- Mapping types
- Other types (e.g., set types and callable, class, module, user-defined object types)

TODO: Probably should elaborate the “other types” more than currently.

2.4.1 Singleton types

Python 3 has single-element types used for special purposes.

2.4.1.1 `None`

The name `None` denotes a value of a singleton type. That is, the type has one element written as `None`.

Python programs normally use `None` to mean there is no meaningful value of another type.

2.4.1.2 `NotImplemented`

The name `NotImplemented` also denotes a value of a singleton type. Python programs normally ruse his value to mean that an arithmetic or comparison operation is not implemented.

2.4.2 Number types

Core Python 3 supports four types of numbers:

- integers
- real numbers
- complex numbers
- Booleans

2.4.2.1 Integers (`int`)

Type `int` denotes the set of integers. They are encoded in a variant of two's complement binary numbers in the underlyng hardware. They are of unbounded precision, but they are, of course, limited in size by the available virtual memory.

```
>>> type(1)
<class 'int'>
>>> type(-14)
<class 'int'>
>>> x = 2
>>> type(x)
<class 'int'>
```

2.4.2.2 Real numbers (`float`)

Type `float` denotes the subset of the real numbers that can be encoded as double precision floating point numbers in the underlying hardware.

```
>>> type(1.01)
<class 'float'>
>>> type(-14.3)
<class 'float'>
>>> x = 2
>>> type(x)
<class 'int'>
>>> y = 2.0
>>> type(y)
<class 'float'>
>>> x == y  # Note result of equality comparison
True
```

2.4.2.3 Complex numbers (`complex`)

Type `complex` denotes a subset of the complex numbers encoded as a pair of floats, one for the real part and one for the imaginary part.

```
>>> type(complex('1+2j')) # real part 1, imaginary part 2
<class 'complex'>
>>> complex('1') == 1.0   # Note result of comparison
True
```

```
>>> complex('1') == 1      # Note result of comparison
True
```

2.4.2.4 Booleans (`bool`)

Type `bool` denotes the set of Boolean values `False` and `True`; in Python, this is a subtype of `int` with `False` and `True` having the values 0 and 1, respectively.

```
>>> type(False)
<class 'bool'>
>>> type(True)
<class 'bool'>
>>> True == 1
True
```

Making `bool` a subtype of `int` is an unfortunate legacy design choice from the early days of Python. It is better not to rely on this feature in Python 3 programs.

2.4.2.5 Truthy and falsy values

Python 3 programs can test any object as if it was a Boolean (e.g. within the condition of an `if` or `while` statement or as an operand of a Boolean operation).

An object is *falsy* (i.e. considered as `False`) if its class defines

- a special method `__bool__()` that, when called with the object, returns `False`
- a special method `__len__()` that returns `0`

Note: We discuss special methods in a later section.

Otherwise, the object is *truthy* (i.e. considered as `True`).

The singleton value `NotImplemented` is explicitly defined as truthy.

Falsy built-in values include:

- constants `False` and `None`
- numeric values of zero such as `0`, `0.0`, and `0j`
- empty sequences and collections such as `''`, `()`, `[]`, and `{}` (defined below)

Unless otherwise documented, any function expected to return a Boolean result should return `False` or `0` for false and `True` or `1` for true. However, the Boolean operations `or` and `and` should always return one of their operands.

2.4.3 Sequence types

A *sequence* denotes a serially ordered collection of zero or more objects. An object may occur more than once in a sequence.

Python 3 supports a number of core sequence types. Some sequences have immutable structures and some have mutable.

2.4.3.1 Immutable sequences

An immutable sequence is a sequence in which the structure cannot be changed after initialization.

2.4.3.1.1 `str`

Type `str` (string) denotes sequences of text characters – that is, of Unicode code points in Python 3. We can express strings syntactically by putting the characters between single, double, or triple quotes. The latter supports multi-line strings.

Python does not have a separate character type; a characer is a single-element `str`.

```
>>> type('Hello world')
<class 'str'>
>>> type("Hi Earth")
<class 'str'>
>>> type('''
... Can have embedded newlines
... ''')
<class 'str'>
```

2.4.3.1.2 `tuple`

Type `tuple` denotes fixed length, heterogeneous sequences of objects. We can express tuples syntactically as sequences of comma-separated expressions in parentheses.

The tuple itself is immutable, but the objects in the sequence might be mutable.

```
>>> type(())    # empty tuple
<class 'tuple'>
>>> type((1,))  # one-element tuple
<class 'tuple'>
>>> x = (1,'Ole Miss')  # mixed elements
>>> type(x)
<class 'tuple'>
>>> x[0]        # access element with index 0
```

```
1
>>> x[1]        # access element with index 1
'Ole Miss'
```

2.4.3.1.3 `bytes`

Type `bytes` denotes sequences of 8-bit bytes. We can express these syntactically as ASCII character strings prefixed by a "`b`".

```
>>> type(b'Hello\n   World!')
<class 'bytes>
```

2.4.3.2 Mutable sequences

A mutable sequence is a sequence in which the structure can be changed after initialization.

2.4.3.2.1 `list`

Type `list` denotes variable-length, heterogeneous sequences of objects. We can express lists syntactically as comma-separated sequence of expressions between square brackets.

```
>>> type([])
<class 'list'>
>>> type([3])
<class 'list'>
>>> x = [1,2,3] + ['four','five'] # concatenation
>>> x
[1, 2, 3, 'four', 'five']
>>> type(x)
<class 'list'>
>>> y = x[1:3] # get slice of list
>>> y
[2, 3]
>>> y[0] = 3   # assign to list index 0
[3, 3]
```

2.4.3.2.2 `bytearray`

Type `bytearray` denotes mutable sequences of 8-bit bytes, that is otherwise like type `bytes`. They are constructed by calling the function `bytearray()`.

```
>>> type(bytearray(b'Hello\n   World!'))
<class 'bytes>
```

2.4.4 Mapping types

Type `dict` (dictionary) denotes mutable finite sets of key-value pairs, where the *key* is an index into the set for the value with which it is paired.

The key must be an immutable object to enable use of hashing. However, the associated value objects may be mutable and the membership in the set may change.

We can express dictionaries syntactically in various ways such as comma-separated lists of key-value pairs with braces.

```
>>> x = { 1 : "one" }
>>> x
{1: 'one'}
>>> type(x)
<class 'dict'>
>>> x[1]
'one'
>>> x.update({ 2 : "two" }) # add to dictionary
>>> x
{1: 'one', 2: 'two'}
>>> type(x)
<class 'dict'>
>>> del x[1]  # delete element with key
>>> x
{2: 'two'}
```

2.4.5 Other object types

We discuss callable objects (e.g. functions), class objects, module objects, and user-defined types (classes) below.

We do not discuss the set types here.

TODO: Probably should add some discussion of sets here

2.5 Statements

The basic building blocks of Python 3 programs include statements, functions, classes, and modules. We discuss those in this and the following subsections.

Python 3 *statements* consist primarily of assignment statements and other mutator operation and constructs to control the order in which those are executed.

Statements execute in the order defined in the program text (as shown below). Each statement executes in an *environment* that assigns values to the names (e.g., of variables and functions) that occur in the statement.

```
statement1
statement2
statement3
...
```

TODO: Do I need to say more here about kinds of statements? Give a reference? an example?

2.6 Functions

Python 3 *functions* are program units that take zero or more *arguments* and *return* a corresponding value.

The code below shows the general structure of a function definition.

```
def my_func(x, y, z):
    statement1
    statement2
    statement3
    return my_loc_var
```

If a function does not explicitly return a value, it implicitly returns the singleton object `None`.

When a program *calls* a function, it passes a *reference* (pointer) to each *argument* object. These references are *bound* to the corresponding *parameter* names, which are *local* variables of the function.

If we assign a new object to the parameter variable in the called function, then the variable binds to the new object. This new binding is *not visible* to the calling program.

However, if we apply a mutator or destructor to the parameter and the argument object is mutable, we can modify the actual argument object. The modified value is *visible* to the calling program.

Of course, if the argument object is not mutable, we cannot modify it's value.

Functions in Python 3 are *first-class objects*. That is, they are (callable) objects of type `function` and, hence, can be stored in data structures, passed as arguments to functions, and returned as the value of a functions. Like other objects, they can have associated data attributes.

To see this, consider the function `add2` and the following series of commands in the Python REPL.

```
>>> def add3(x, y, z):
...     return x + y + z
...
>>> add3(1,2,3)
```

```
6
>>> type(add3)
<class `function`>
>>> x = [add3,1,2,3,6]  # store function object in list
>>> x
[<function add3 at 0x10bf65ea0>, 1, 2, 3, 6]
>>> add3.author = 'Cunningham'  # set attribute author
>>> add3.author                 # get attribute author
'Cunningham'
```

We call a function a *higher-order function* if it takes another function as its parameter and/or returns a function as its return value.

2.7 Classes

A Python 3 *class* is a program construct that defines a new nominal *type* consisting of data attributes and the operations on them. When we call a class as a function, it creates a new *instance* (i.e. an object) of the associated type.

We define an operation with a method bound to the class. A *method* is a function that takes an instance (by convention named `self`) as its first argument. It can access and modify the data attributes of the instance. The method is also an attribute of the instance.

The code below shows the general structure of a class definition. The class calls the special method `__init__` (if present) to initialize a newly allocated instance of the class.

Note: The special method `__new__` allocates memory, constructs a new instance, and then returns it. The interpretrer passes the new instance to `__init__`. which initizlizes the new object's instance variables.

```
class P:
    def __init__(self):
        self.my_loc_var = None
    def method1(self, args):
        statement11
        statement12
        return some_value
    def method2(self, args):
        statement21
        statement22
        return some_other_value
```

A class instance defines an environment with local variable and method names and their values.

In addition to being factories for creating and initializing instances, Python 3 classes are themselves objects.

Consider the following simple example.

```
class P:
    pass

>>> x = P()
>>> x
<__main__.P object at 0x1011a10b8>
>>> type(x)
<class '__main__.P'>
>>> isinstance(x,P)
True
>>> P
<class '__main__.P'>
>>> type(P)
<class 'type'>
>>> isinstance(P,type)
True
>>> int
<class 'int'>
>>> type(int)
<class 'type'>
>>> isinstance(int,type)
True
```

We observe the following:

- Variable `x` holds a value that is an object of type `P`; the object is an instance of class `P`.
- Class `P` is an object of a built-in type named `type`; the object is an instance of class `type`.
- Built-in type `int` is also an object of the type named `type`.

We call a class object like `P` a *metaobject* because it is a constructor of ordinary objects [Kiczales 1991; Forman 1999].

We call a special class object like `type` a *metaclass* because it is a constructor for metaobjects (i.e., class objects) [Kiczales 1991; Forman 1999].

We will look more deeply into these relationships later when we examine inheritance.

2.8 Modules

A Python 3 *module* is a file that contains a sequence of *global* variable, function, and class definitions and executable statements. If the name of the file is `mymod.py`, then the module's name is `mymod`.

A Python 3 *package* is a directory of Python 3 modules.

A module collects the names and values of its global variables, functions, and classes into its own private *namespace* (i.e. environment). This becomes the global environment for all definitions and executable statements in the module.

When we execute a module as a script from the Python 3 REPL, the interpreter executes all the top-level statements in the module's namespace. If the module contains function or class definitions, then the interpreter checks those for syntactic correctness and stores the definitions in the namespace for use later during execution.

2.8.1 Using `import`

Suppose we have the following Python 3 code in a file named `test.py`.

```
# This is module "testmod" in file "testmod.py"
testvar = -1

def test(x):
    return x
```

We can execute this code in a Python 3 REPL session as follows.

```
>>> import testmod   # import module in file "testmod.py"
>>> testmod.testvar  # access module's variable "testvar"
-1
>>> testmod.testvar = -2  # set variable to new value
>>> testmod.testvar
-2
>>> testmod.test(23) # call module's function "test"
23
>>> test(2)          # must use module prefix "test"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'module' object is not callable
>>> testmod          # below PATH = directory path
<module 'testmod' from 'PATH/testmod.py'>
>>> type(testmod)
<class 'module'>
>>> testmod.__name__
'testmod'
```

```
>>> type(type(testmod))
<class 'type'>
```

The `import` statement causes the interpreter to execute all the top-level statements from the module file and makes the namespace available for use in the script or another module. In the above, the imported namespace includes the variable `testvar` and the function definition `test`.

A name from one module (e.g., `testmod`) can be directly accessed from an imported module by prefixing the name by the module name using the dot notation. For example, `testmod.testvar` accesses variable `testvar` in module `testmod` and `testmod.test()` calls function `test` in module `testmod`.

We also see that the imported module `testmod` is an object of type (class) `module`.

2.8.2 Using `from import`

We can also *import* names selectively. In this case, the definitions of the selected features are copied into the module.

Consider the module `testimp` below.

```
# This is module "testimp" in file "testimp.py"
from testmod import testvar, test

myvar = 10

def myfun(x, y, z):
    mylocvar = myvar + testvar
    return mylocvar

class P:
    def __init__(self):
        self.my_loc_var = None

    def meth1(self, arg):
        return test(arg)

    def meth2(self, arg):
        if arg == None:
            return None
        else:
            my_loc_var= arg
            return arg
```

The definitions of variable `testvar` and function `test` are copied from module `testmod` into module `testimp`'s namespace. Module `testimp` can thus access

these without prefix `testmod`.

Module `testimp` could import all of the definitions from module `testmod` by using the wildcard `*` instead of the explicit list.

We can execute the above code in a Python 3 REPL session as follows.

```
>>> import testimp
>>> testimp.myvar
10
>>> testimp.myfun(1,2,3)
9
>>> pp = testimp.P()
>>> pp.meth1(23)
23
>>> pp.meth2(14)
14
>>> type(pp)
<class 'testimp.P'>
>>> type(testimp.testmod)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'testmod' is not defined
```

Note that the `from testmod import` statement does not create an object `testmod`.

2.8.3 Programming conventions

Python 3 programs typically observe the following conventions:

- All module `import` and `import from` statements should appear at the beginning of the importing module.
- All `from import` statements should specify the imported names explicitly rather than using the wildcard `*` to import all names. This avoids polluting the importing module's namespace with unneeded names. It also makes the dependencies explicit.
- Any definition whose name begins with an `_` (underscore) should be kept private to a module and thus should not be imported into or accessed directly from other modules.

2.8.4 Using `importlib` directly

TODO: Perhaps move the discussion below of the `importlib`, a metaprogramming feature, to a later chapter.

The Python 3 core module `importlib` exposes the functionality underlying the `import` statement to Python 3 programs. In particular, we can use the function call

```
importlib.import_module('modname') # argument is string
```

to find and import a module from the file named `modname.py`. Below we see that this works like an explicit `import`.

```
>>> from importlib import import_module
>>> tm = import_module('testmod')
>>> tm          # below PATH = directory path
<module 'testmod' from 'PATH/testmod.py'>
>>> type(tm)
<class 'module'>
>>> type(type(tm))
<class 'type'>
```

2.9 Statement Execution and Variable Scope

Statements perform the work of the program – computing the values of expressions and assigning the computed values to variables or parts of data structures.

Statements execute in two scopes: global and local.

1. As described above, the *global* scope is the enclosing module's environment (a dictionary), as extended by imports of other modules.
2. As described above, the *local* scope is the enclosing function's dictionary (if the statement is in a function).

   If `statement` is a string holding a Python 3 statement, then we can execute the statement dynamically using the `exec` library function. By default, the statement is executed in the current global and local environment, but these environments can be passed in explicitly.

   `exec(statement` [`,` globals [`,` locals] `)`

Inside a function, variables that are:

- referenced but not assigned a value are assumed to be global
- assigned a value are assumed to be local

In the latter case, we can explicitly declare the variable `global`. if the desired target variable is defined in the global scope.

2.10 Function Calling Conventions

Consider a module-level function. A function may include a combination of:

- positional parameters
- keyword parameters

There are several different ways we can specify the arguments of function calls described below.

1. Using *positional arguments*

```
def myfunc(x, y, z):
    statement1
    statement2
    ...
myfunc(10, 20, 30)
```

2. Using *keyword* arguments

```
def myfunc(x, y, z):
    statement1
    statement2
    ...
myfunc(z=30, x=10, y=20)
# note different order than in signature
```

3. Using *default arguments* set at definition time – using only immutable values (e.g., `False`, `None`, string, tuple) for defaults

```
def myfunc(x, trace = False, vars = None):
    if vars is None:
        vars = []
    ...
myfunc(10)
    # x=10, trace=False, vars=None
myfunc(10, vars=['x', 'y'])
    # x=10, trace=False, vars=['x', 'y'])
```

4. Using required positional and *variadic positional* arguments

```
def myfunc(x, *args):
    # x is a required argument in position 1
    # args is tuple of variadic positional args
    # name "args" is just convention
    ...
myfunc(10, 20, 30)
    # x = 10
    # args = (20, 30)
```

5. Using required positional, variadic positional, and keyword arguments

```
def myfunc(x, *args, y):
    # x is a required argument in position 1
    # args is tuple of variadic positional args
```

```
    # y is keyword argument (occurs after variadic positional)
    ...
myfunc(10, 20, 30, y = 40)
    # x = 10
    # args = (20, 30)
    # y = 40
```

6. Using required positional, variadic positional, keyword, and *variadic keyword* arguments

```
def myfunc(x, *args, y = 40, **kwargs):
    # x is a required argument in position 1
    # args is tuple of variadic positional args
    # y is a regular keyword argument with default
    # kwargs is a dictionary of variadic keyword args
    # names 'args' and 'kwargs' are conventions
    ...
myfunc(10, 20, 30, y = 40, r = 50, s = 60, t = 70)
    # x = 10
    # args = (20, 30)
    # y = 40
    # kwargs = { 'r': 50, 's': 60, 't': 70 }
```

7. Using required positional and keyword arguments – where named arguments appearing after * can only be passed by keyword

```
def myfunc(x, *, y, **kwargs):
    # x is a required argument in position 1
    # y is a regular keyword argument
    # kwargs is a dictionary of keyword args
   ...
myfunc(10, y = 40, r = 50, s = 60, t = 70)
    # x = 10
    # y = 40
    # kwargs = { 'r': 50, 's': 60, 't': 70 }
```

8. Using a fully variadic general signature

```
def myfunc(*args, **kwargs):
    # args is tuple of all positional args
    # kwargs is a dictionary of all keyword args
    ...
myfunc(10, 20, y = 40, 30, r = 50, s = 60, t = 70)
    # args = (10, 20, 30)
    # kwargs = { 'y': 40, 'r': 50, 's': 60, 't': 70 }
```

2.11 Nested Function Definitions

Above we only considered module-level function definitions and instance method definitions defined within classes.

Python 3 allows function definitions to be nested within other function definitions. Nested functions have several characteristics:

- *Encapsulation.* The outer function hides the inner function definitions from the global scope. The inner functions can only be called from within the outer function.

  In contrast, Python 3 classes and modules do not provide airtight encapsulation. Their hiding of information is mostly by convention, with some support from the language.

- *Abstraction.* The inner function is a procedural abstraction that is named and separated from the outer function's code. This enables the inner function to be used several times within the outer function. The abstraction can enable the algorithm to be simplified and understood more easily.

  Of course, modules and classes also support abstraction, but not in combination with encapsulation.

- *Closure construction.* The outer function can take one or more functions as arguments, combine them in various ways (perhaps with inner function definitions), and construct and return a specialized function as a *closure*. The closure can bind in parameters and other local variables of the outer function.

  Closures enable functional programming techniques such as currying, partial evaluation, function composition, construction of combinators, etc.

  We discuss closures in more depth in a later section of this chapter.

  Closure are powerful mechanisms that can be used to implement metaprogramming solutions (e.g., Python 3's decorators). We discuss those in later chapters.

As an example of use of nested function definitions to promote encapsulation and abstraction, consider a recursive function `sqrt(x)` to compute the square root of nonnegative number `x` using Newton's Method. (This is adapted from section 1.1.7 of [Abelson 1996].)

```
def sqrt(x):
    def square(x):
        return x * x
    def good_enough(guess,x):
        return abs(square(guess) - x) < 0.001
    def average(x,y):
        return (x + y) / 2
```

```
def improve(guess,x):
    return average(guess,x/guess)
def sqrt_iter(guess,x):  # recursive version
    if good_enough(guess,x):
        return guess
    else:
        return sqrt_iter(improve(guess,x),x)
if x >= 0:
    return sqrt_iter(1, x)
else:
    print(
        f'Cannot compute square root of negative number {x}')
```

A more "Pythonic" implementation of the `sqrt_iter` function would use a loop as follows:

```
def sqrt_iter(guess,x):  # looping version
    while not good_enough(guess,x):
        guess = improve(guess,x)
    return guess
```

Note: The Python 3.6+ source code for the recursive version of `sqrt` is available at this link and the looping version at another link.

2.12 Lexical Scope

Nested function definitions introduce a third category of variables – local variables of outer functions – in addition to the (function-level) local and (module-level) global scopes we have discussed so far.

Python 3 searches *lexical scope* (also called *static scope*) of a function for variable accesses.

Inside a function, variables that are:

- referenced but not assigned a value are assumed to be either defined in an outer function scope or in the global scope.

  The Python 3 interpreter first searches for the nearest enclosing function scope with a definition. If there is none, it then searches the global scope.

- assigned a value are assumed to be local

In the latter case, we can explicitly declare the variable as `nonlocal` if the desired variable to be assigned is defined in an enclosing function scope or as `global` if it is defined in the global scope.

Suppose we want to add an iteration counter `c` to the `sqrt` function above. We can create and initialize variable `c` in the outer function `sqrt`, but we must increment it in nested function `sqrt_iter`. For the nested function to change an

outer function variable, we must declare the variable as `nonlocal` in the nested function's scope.

```
def sqrt(x):
    c = 0                   # create c in outer function
    # same definitions of square, good_enough, average, improve
    def sqrt_iter(guess,x): # new local x, hide outer x
        nonlocal c          # declare c nonlocal
        while not good_enough(guess,x):
            c += 1          # increment c
            guess = improve(guess,x)
        return (guess,c)    # return c
    if x >= 0:
        return sqrt_iter(1, x)
    else:
        print(f'Cannot compute square root of negative number {x}')
```

Note: The Python 3.6+ source code for this version of `sqrt` is available at this link.

2.13 Closures

As discussed in a previous section, Python 3 function definitions can be nested inside other functions. Among other capabilities, this enables a Python 3 function to create and return a closure.

A *closure* is a function object plus a reference to the enclosing environment.

For example, consider the following:

```
def make_multiplier(x, y):
    def mul():
       return x * y
    return mul
```

If we call this function interactively from the Python 3 REPL, we see that the values of the local variables `x` and `y` are captured by the function returned.

```
>>> amul = make_multiplier(2, 3)
>>> bmul = make_multiplier(10, 20)
>>> type(amul)
<class 'function'>
>>> amul()
6
>>> bmul()
200
```

Function `make_multiplier` is a *higher order function* because it returns a function (or closure) as its return value. Higher order functions may also take

functions (or closures) as parameters.

We can compose two conforming single argument functions using the following `compose2` function. Function `comp` captures the two arguments of `compose2` in a closure [Larose 2013].

```
def compose2(f, g):
    def comp(x):
        return f(g(x))
    return comp
```

Given that `f(g(x))` is a simple expression without side effects, we can replace the `comp` function with an anonymous `lambda` function as follows:

```
def compose2(f, g):
    return lambda x: f(g(x))
```

If we call this function from the Python 3 REPL, we see that the values of the local variables `x` and `y` are captured by the function returned.

```
>>> def square(x):
...     return x * x
...
>>> def inc(x):
...     return x + 1
...
>>> inc_then_square = compose2(square, inc)
>>> inc_then_square(10)
121
```

Note: The Python 3.6+ source code for `compose2` is available at this link.

2.14 Class and Instance Attributes

As we noted above, classes are objects. The class objects can have attributes. Instances of the class are also objects with their own attributes.

Consider the following class `Dummy` which has a class-level variable `r`. This attribute exists even if no instance has been created.

Instances of `Dummy` have instance variables `s` and `t` and an instance method `in_meth`.

```
class Dummy:
    r = 1
    def __init__(self, s, t):
        self.s = s
        self.t = t
   def in_meth(self):
        print('In instance method in_meth')
```

Now consider the following Python 3 REPL session with the above definition.

```
>>> Dummy.r
1
>>> d = Dummy(2,3)
>>> d.s
2
>>> d.in_meth()
>>> In instance method method
```

In the above, we see that:

- `Dummy.r` accesses the value of class variable `r` of the class object for the class `Dummy`.
- `d.s` accesses the value of instance variable `s` of an instance object created by the constructor call and assignment `d = Dummy(2)`.
- `d.in_meth()` calls instance method `in_meth` of instance object `d`.

These usages are similar to those of other object-oriented languages such as Java.

A class can have three different kinds of methods in Python 3 [StackOverflow 2012]:

1. An *instance method* is a function associated with an instance of the class. It requires a reference to an instance object to be passed as the first non-optional argument, which is by convention named `self`. If that reference is missing, the call results in a `TypeError`.

   It can access the values of any of the instance's attributes (via the `self` argument) as well as the class's attributes.

   Note `in_meth` in the `Dummy` code below.

2. A *class method* is a function associated with a class object. It requires a reference to the class object to be passed as the first non-optional argument, which is by convention named `cls`. If that reference is missing, the call results in a `TypeError`.

   It can access the values of any of the class's attributes (via the `cls` argument). For example, `cls()` can create a new instance of the class. However, it cannot access the attributes of any of the class's instances.

   Note `cl_meth` in the `Dummy` code below.

   Class methods can be overriden in subclasses.

   Because Python 3 does not support method overloading, class methods are useful in circumstances where overloading might be used in a language like Java. For example, we can use class methods to implement factory methods as alternative constructors for instances of the class.

3. A *static method* is a function associated with the class object, but it does not require any non-optional argument to be passed

   A static method is just a function attached to the class's namespace. It cannot access any of the attributes of the class or instances except in a way that any function in the program can (e.g., by using the name of the class explicitly, by being passed an object as an argument, etc.)

   Note `s_meth` in the `Dummy` code below.

   Static methods cannot be overrides in subclasses.

```
class Dummy:  # extended definition
    r = 1

    def __init__(self, s, t):
        self.s = s
        self.t = t

    def in_meth(self):
        print('In instance method in_meth')

    @classmethod
    def cl_meth(cls):
        print(f'In class method cl_meth for {cls}')

    @staticmethod
    def st_meth():
        print('In static method st_meth')
```

In the example, the *decorators* `@classmethod` and `@staticmethod` transform the attached functions into class and static methods, respectively. We will discuss decorators in a later section.

Now consider a Python 3 REPL session with the extended definition.

```
>>> d = Dummy(2,3)
>>> d.in_meth()
In instance method in_meth
>>> d.r
1
>>> Dummy.cl_meth()
In class method cl_meth for Dummy
>>> Dummy.st_meth()
In static method st_meth
>>> Dummy.in_meth()
Traceback (most recent call last):
...
TypeError: in_meth() missing 1 required positional argument: 'self'
```

```
>>> type(d.in_meth)
<class 'method'>
>>> type(Dummy.cl_meth)
<class 'method'>
>>> type(Dummy.st_meth)
<class 'function'>
>>> din = d.in_meth  # get method object, store in variable din
>>> din()            # call method object in din
In instance method in_meth
None
>>> type(din)
<class 'method'>
```

Note that the types of the references `d.in_meth` and `Dummy.cl_meth` are both `method`. A `method` object is essentially a function that binds in a reference to the required first positional argument. A method object is, of course, a first-class object that can be stored and invoked later as illustrated by variable `din` above.

However, note that `Dummy.st_meth` has type `function`.

Note: The Python 3.6+ source code for the class `Dummy` is available at this link.

2.15 Object Dictionaries

As we noted earlier, each Python 3 object has a distinct dictionary that maps the local names to the data attributes and operations (i.e., its environment). Each object's attribute `__dict__` holds its dictionary. Python 3 programs can access this dictionary directly.

Again consider the `Dummy` class we examined the previous section. Let's look at dictionary for this class and an instance in the Python 3 REPL.

```
>>> d = Dummy(2,3)
>>> d.__dict__
{'s': 2, 't': 3}
>>> Dummy.__dict__["r"]
1
>>> Dummy.__dict__["in_meth"]
<function Dummy.in_meth at 0x10191abf8>
>>> Dummy.__dict__["cl_meth"]
<classmethod object at 0x101928c50>
>>> Dummy.__dict__["st_meth"]
<staticmethod object at 0x101928c88>
```

TODO: Investigate and explain last two types returned above?

2.16 Special Methods and Operator Overloading

Almost everything about the behavior of Python 3 classes and instances can be customized. A key way to do this is by defining or redefining *special methods* (sometimes called *magic* methods).

Python 3 uses special methods to provide an *operator overloading* capability. There are special methods associated with certain operations that are invoked by builtin operators (such as arithmetic and comparison operators, subscripting) and with other functionality (such as initializing newly class instance).

The names of special methods both begin and end with double underscores `__` (and thus are sometimes called "dunder" methods). For example, in an earlier subsection, we defined the special method `__init__` to specify how a newly created instance is initialized. In other class-based examples, we have defined the `__str__` special method to implement a custom string conversion for an instance's state.

Consider the class `Dum` that overrides the definition of the addition operator to do the same operation as multiplication.

```
class Dum:
    def __init__(self,x):
       self.x = x
    def __add__(a,b):
        return a.x * b.x
```

Now let's see how `Dum` works.

```
>>> y = Dum(2)
>>> z = Dum(4)
>>> y + z
8
```

Consider the rudimentary `SparseArray` collection below. It uses the special methods `__init__`, `__str__`, `__getitem__`, `__setitem__`, `__delitem__`, and `__contains__` to tie this new collection into the standard access mechanisms. (This example stores the sparse array in a dictionary internally, but "hides" that from the user.)

The Boolean `__contains__` functionality searches the `SparseArray` instance for an item. The class also provides a separate Boolean method `has_index` to check wether an index has a corresponding value. Alternatively, we could have tied the `__contains__` functionality to the latter and provided a `has_item` method for the former.

In addition, the method `from_assoc` loads an "association list" into a sparse array instance. Here, the term *association list* refers to any iterable object yielding a finite sequence of index-value pairs.

Similarly, the method `to_assoc` unloads the entire sparse array into a sorted list of index-value pairs (which is an iterable object).

For simplicity, the implementation below just prints error messages. It probably should raise exception instead.

```
class SparseArray:

    def __init__(self, assoc=None):
        self._arr = {}
        if assoc is not None:
            self.from_assoc(assoc)

    def from_assoc(self,assoc):
        for p in assoc:
            if len(p) == 2:
                (i,v) = p
                if type(i) is int:
                    self._arr[i] = v
                else:
                    print(
                        f'Index not int in assoc list: {str(i)}')
            else:
                print(f'Invalid pair in assoc list: {str(p)}')

    def has_index(self, index):
        if type(index) is int:
            return index in self._arr
        else:
            print(f'Warning: Index not int: {index}')
            return False

    def __getitem__(self, index):        # arr[index]
        if type(index) is int:
            return self._arr[index]
        else:
            print(f'Index not int: {index}')

    def __setitem__(self, index, value): # arr[index] = value
        if type(index) is int:
            self._arr[index] = value
        else:
            print(f'Index not int: {index}')

    def __delitem__(self, index):        # del arr[index]
        if type(index) is int:
            del self._arr[index]
```

```
            else:
                print(f'Index not int: {index}')

        def __contains__(self, item):        # item (is value) in arr
            return item in self._arr.values()

        def to_assoc(self):
            return sorted(self._arr.items())

        def __str__(self):
            return str(self.to_assoc())
```

Now consider a Python 3 REPL session with the above class definition.

```
>>> arr = SparseArray()
>>> type(arr)
<class '__main__.SparseArray'>
>>> arr
[]
>>> arr[1] = "one"
>>> arr
[(1, `one`)]
>>> arr.has_index(1)
True
>>> arr.has_index(2)
False
>>> arr.from_assoc([(2,"two"),(3,"three")])
{1: 'one', 2: 'two', 3: 'three'}
>>>
>>> arr[10] = "ten"
>>> arr
[(1,'one'), (2, 'two'), (3, 'three'), (10, 'ten')]
>>> del arr[3]
>>> arr
[(1,'one'), (2, 'two'), (10, 'ten')]
>>> 'ten' in arr
True
```

Note: The Python 3.6+ source code for the class `SparseArray` is available at this line.

2.17 Object Orientation

The Programming Paradigms notes [Cunningham 2018a] discuss object orientation in terms of a general object model. The general *object model* includes four basic components:

1. objects
2. classes
3. inheritance
4. subtype polymorphism

We discuss Python 3's objects and classes above. Now let's consider the other two components of the general object model in relation to Python 3.

2.17.1 Inheritance

In programming languages in general, inheritance involves defining hierarchical relationships among classes. From a *pure* perspective, a class `C` *inherits* from class `P` if `C`'s objects form a *subset* of `P`'s objects in the following sense:

- Class `C`'s objects must support all of class `P`'s operations (but perhaps are carried out in a special way).

  We can say that a `C` object *is a* `P` object or that a class `C` object can be *substituted* for a class `P` object whenever the latter is required.

- Class `C` may support additional operations and an extended state (i.e., more data attributes fields).

We use the following terminology.

- Class `C` is called a *subclass* or a *child* or *derived class*.
- Class `P` is called a *superclass* or a *parent* or *base* class.
- Class `P` is sometimes called a *generalization* of class `C`; class `C` is a *specialization* of class `P`.

In terms of the discussion in the Type System Concepts section, the parent class `P` defines a conceptual type and child class `C` defines a behavioral subtype of `P`'s type. The subtype satisfies the Liskov Substitution Principle.

Even in a statically typed language like Java, the language does not enforce this subtype relationship. It is possible to create subclasses that are not subtypes. However, using inheritance to define subtype relationships is considered good object-oriented programming practice in most circumstances.

In a dynamically typed like Python 3, there are fewer supports than in statically typed languages. But using classes to define subtype relationships is still a good practice.

The importance of inheritance is that it encourages sharing and reuse of both design information and program code. The shared state and operations can be described and implemented in parent classes and shared among the child classes.

The following code fragment shows how to define a single inheritance relationship among classes in Python 3. Instance method `process` is defined in the parent

class `P` and *overridden* (i.e., redefined) in child class `C` but not overriden in child class `D`. In turn, `C`'s instance method `process` is overridden in its child class `G`.

```
class P:
    def __init__(self,name=None):
        self.name = name
    def process(self):
        return f'Process at parent P level'

class C(P):   # class C inherits from class P
    def process(self):
        result = f'Process at child C level'
        # Call method in parent class
        return f'{result} \n  {super().process()}'

class D(P):   # class D inherits from class P
    pass

class G(C):   # class G inherits from class C
    def process(self):
        return f'Process at grandchild G level'
```

Now consider a (lengthy) Python 3 REPL session with the above class definition.

```
>>> p1 = P()
>>> c1 = C()
>>> d1 = D()
>>> g1 = G()
>>> p1.process()
'Process at parent P level'
>>> c1.process()
'Process at child C level'
'Process at parent P level'
>>> d1.process()
'Process at parent P level'
>>> g1.process()
'Process at grandchild G level'
#
>>> type(P)
<class 'type'>
>>> type(C)
<class 'type'>
>>> type(G)
<class 'type'>
>>> issubclass(P,object)
True
>>> issubclass(C,P)
```

```
True
>>> issubclass(G,C)
True
>>> issubclass(G,P)
True
>>> issubclass(G,object)
True
>>> issubclass(C,G)
False
>>> issubclass(G,D)
False
>>> issubclass(P,type)
False
>>> isinstance(P,type)
True
>>> isinstance(C,type)
True
>>> isinstance(G,type)
True
>>> type(type)
<class 'type'>
>>> issubclass(type,object)
True
>>> isinstance(type,type)
True
>>> type(object)
<class 'type'>
>>> isinstance(object,type)
True
>>> issubclass(object,type)
False
```

`Note: The Python 3.6+ source code for the above version of the`P‘ class hierarchy is available at this link.

2.17.1.1 Understanding relationships among classes

By examining the REPL session above, we can observe the following:

- Top-level user-defined classes like `P` implicitly inherit from the `object` root class. They have the `issubclass` relationship with `object`.
- A user-defined subclass like `C` inherits explicitly from its superclass `P`, which inherits implicitly from root class `object`. Class `C` thus has `issubclass` relationships with both `P` and `object`.
- By default, all Python 3 classes (including subclasses) are instances of the

root metaclass `type` (or one of its subtypes as we see later). But non-class objects are not instances of `type`.

As we noted in a previous section, we call class objects *metaobjects*; they are constructors for ordinary objects [Kiczales 1991; Forman 1999].

Also as we noted in a previous section, we call special class objects like `type` *metaclasses*; they are constructors for metaobjects (i.e., class objects) [Kiczales 1991; Forman 1999].

Note that classes `object` and `type` have special – almost "magical" – relationships with one another [Ramalho 2015, pp. 593-5].

- Class `object` is an instance of class `type` (i.e. it is a Python class object).
- Class `type` is an instance of itself (i.e. it is a Python class object) and a subclass of class `object`.

The diagram in Figure 1 shows the relationships among user-definfed class `P` and built-in classes `int`, `object`, and `type`. Solid lines denote subclass relationships; dashed lines denote "instance of" relationships.

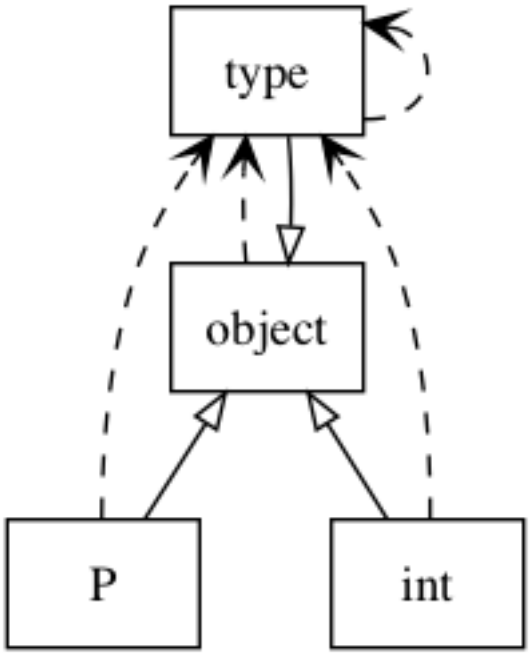

Figure 1: Python 3 Class Model

2.17.1.2 Replacement and refinement

There are two general approaches for overriding methods in subclasses.

- *Replacement*, in which the child class method totally replaces the parent class method

  This is the usual approach in most "American school" object-oriented languages in use today – Smalltalk (where it originated), Java, C++, C#, and Python 3.

- *Refinement*, in which the language merges the behaviors of the parent and child classes to form a new behavior

  This is the approach taken in Simula 67 (the first object-oriented language) and its successors in the "Scandinavian school" of object-oriented languages. In these languages, the child class method typically wraps around a call to the parent class method.

The refinement approach supports the implementation of pure subtyping relationships better than replacement does. The replacement approach is more flexible than refinement.

A language that takes the replacement approach usually provides a mechanism for using refinement. For example in the Python 3 class hierarchy example above, the expression `super().process()` in subclass `C` calls the `process` method of its superclass `P`.

2.17.2 Subtype polymorphism

The concept of *polymorphism* (literally "many forms") means the ability to hide different implementations behind a common interface. Polymorphism appears in several forms in programming languages. Here we examine one form.

In the Python 3 class hierarchy example above, the method `process` forms part of the common interface for this hierarchy. Parent class `P` defines the method, child class `C` overrides `P`'s definition by refinement, and grandchild class `G` overrides `C`'s definition by replacement. However, child class `D` does not override `P`'s definition.

Subtype polymorphism (sometimes called *polymorphism by inheritance*, *inclusion polymorphism*, or *subtyping*) means the association of an operation invocation (e.g., method call) with the appropriate operation implementation in an inheritance (i.e., subtype) hierarchy.

This form of polymorphism is usually carried out at run time. Such an implementation is called *dynamic binding*.

In general, given an object (i.e., class instance) to which an operation is applied, the runtime system first searches for an implementation of the operation associated with the object's class. If no implementation is found, the system checks the parent class, and so forth up the hierarchy until it finds an implementation

and then invokes it. Implementations of the operation may appear at several levels of the hierarchy.

The combination of dynamic binding with a well-chosen inheritance hierarchy allows the possibility of an instance of one subclass being substituted for an instance of a different subclass during execution. Of course, this can only be done when none of the extended operations of the subclass are being used.

In a statically typed language like Java, we declare a variable of some ancestor class type. We can then store any descendant class instance in that variable. Polymorphism allows the program to apply any of the ancestor class operations to the instance.

Because of dynamically typed variables, polymorphism is even more flexible in Python 3 than in Java.

In Python 3, an instance object may also have its own implementation of a method, so the runtime system searches the instance before searching upward in the class hierarchy.

Also (as we noted in an earlier section) Python 3 uses duck typing. Objects can have a common interface even if they do not have common ancestors in a class hierarchy. If the runtime system can find an compatible operation associated with an instance, it can execute it.

Thus Python 3's approach to subtype polymorphism gives considerable flexibility in structuring programs. However, unlike statically typed languages, the compiler provides little help in ensuring the compatibility of method implementations.

Again consider the simple inheritance hierarchy above in the following Python 3 REPL session.

```
>>> d1 = D()
>>> g1 = G()
>>> obj = d1     # variables support polymorphism
>>> obj.process()
'Process at parent P level'
>>> obj = g1     # variables support polymorphism
>>> obj.process()
'Process at grandchild G level'
```

2.17.3 Multiple Inheritance

TODO: Decide whether discussion of multiple inheritance is needed here? or later? Issues include the diamond problem, Python syntax and semantics, and method resolution order.

2.18 Chapter Summary

TODO: Write this

In the next chapter, we explore a case study to motivate the concept and use of decorators and metaclasses in Python 3.

2.19 Exercises

TODO: Decide whether any are needed.

2.20 Acknowledgements

I developed these notes in Spring 2018 for use in CSci 658 Software Language Engineering. The Spring 2018 version used Python 3.6.

The overall set of notes on Python 3 Reflexive Metaprogramming is inspired by David Beazley's Python 3 Metaprogramming tutorial from PyCon'2013 [Beazley 2013a]. In particular, some chapters adapt Beazley's examples. Beazley's tutorial draws on material from his and Brian K. Jones' book *Python Cookbook* [Beazley 2013b].

In particular, this chapter adapts and extends some aspects of the introductory section of [Beazley 2013a].

The notes also adapt some material from the Programming Paradigms [Cunningham 2018a] notes.

I maintain these notes as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the notes to HTML, PDF, and other forms as needed.

2.21 References

[Abelson 1996]: Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs (2nd Edition)*, MIT Press, 1996.

[Beazley 2013a]: David Beazley. Python 3 Metaprogramming (Tutorial), *PyCon'2013*, 14 March 2013.

[Beazley 2013b]: David Beazley and Brian K. Jones. *Python Cookbook, 3rd Edition*, O'Reilly Media, May 2013.

[Cunningham 2018a]: H. Conrad Cunningham. Programming Paradigms, Object-Oriented section, revised 17 February 2018.

[Forman 1999]: Ira R. Forman and Scott Danforth. *Putting Metaclasses to Work*, Addison Wesley Longman, 1999.

[**Kiczales 1991**]: Gregor Kiczales, Jim des Rivieres; Daniel G. Bobrow. *The Art of the Metaobject Protocol*, The MIT Press, 1991.

[**Larose 2013**]: Mathieu Larose. Function Composition in Python, Blog post. January 2013.

[**Liskov 1987**]: Barbara Liskov. Keynote Address – Data Abstraction and Hierarchy, In the Addendum to the *Proceedings on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '87)*, Leigh Power and Zvi Weiss, Editors, ACM, 1987.

[**Ramalho 2015**]: Luciano Ramalho. *Fluent Python: Clear, Concise, and Effective Programming*, O'Reilly Media, May 2015.

[**Scott 2016**]: Michael L. Scott. *Programming Language Pragmatics*, Fourth Edition, Morgan Kaufmann, 2016. MAY NOT BE CITED

[**StackOverflow 2012**]: StackOverflow. Meaning of @classmethod and @staticmethod-for beginner?, first post 12 August 2012.

[**Wikipedia 2018d**]: Wikipedia, Liskov Substitution Principle, accessed 25 April 2018.

2.22 Terms and Concepts

TODO