Multiparadigm Programming with Python Chapter 6

H. Conrad Cunningham

05 April 2022

Contents

Professor of Computer and Information Science
University of Mississippi
214 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-7396 (dept. office)

Browser Advisory: The HTML version of this textbook requires a browser that supports the display of MathML. A good choice as of April 2022 is a recent version of Firefox from Mozilla.

6 Python Program Components

6.1 Chapter Introduction

The basic building blocks of Python programs include statements, functions, classes, and modules. This chapter (6) examines key features of those building blocks.

Chapter 7 examines classes, objects, and object orientation in more detail.

Note: In this book, we use the term Python to mean Python 3. The various examples use Python 3.7 or later.

6.2 Statements

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

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

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

A `statement` may modify the environment by changing the values of variables, creating new variables, reading input, writing output, etc.

A statement may be *simple* or *compound*. We discuss selected simple and compound statements in the following subsections.

6.2.1 Simple statements

A *simple statment* is a statement that does not contain other statements. This subsection examines four simple statements. We discuss other simple statements later in this textbook.

TODO: Make last sentence above more explicit.

6.2.1.1 `pass` statement

The simple statement

```
pass
```

is a null operation. It does nothing. We can use it when the syntax requires a statement but no action is needed.

6.2.1.2 Expression statement An expression statement is a simple statement with the form:

```
expression1, expression2, ...
```

If the expression list has only one element, then the result of the statement is the result of the expression's execution.

If the expression list has two or more elements, then the result of the statement is a sequence (e.g., tuple) of the expressions in the list.

In program scripts, expression statements typically occur where an expression is executed for its side effects (e.g., a procedure call) rather than the value it returns. A procedure call always returns the value `None` to indicate there is no meaningful return value.

However, if called from the Python REPL and the value is not `None`, the REPL converts the result to a string (using built-in function `repr()`) and writes the string to the standard output.

6.2.1.3 Assignment statement A typical Python assignment statement has the form:

```
target1, target2, ... = expression1, expression2, ...
```

The assignment statement evaluates the expression list and generates a sequence object (e.g., tuple). If the target list and the sequence have the same length, the statement assigns the elements of the sequence to the targets left to right. Otherwise, if there is a single target on the left, then the sequence object itself is assigned to the target.

Consider the following REPL session:

```
>>> x, y = 1, 2
>>> x
1
>>> y
2
>>> x = 1, 2, 3
>>> x
(1, 2, 3)
>>> x, y = 1, 2, 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack (expected 2)
```

One of the targets may be prefixed by an asterisk (`*`). In this case, that target *packs* zero or more values into a *list*.

Consider the following REPL session:

```
>>> x, *y, z = 1, 2
>>> y
[]
>>> x, *y, z = 1, 2, 3
>>> y
[2]
>>> x, *y, z = 1, 2, 3, 4, 5
>>> y
[2, 3, 4]
```

Note: See the Python 3 Language Reference manual [10] for a more complete explanation of the syntax and semantics of assignment statements.

TODO: Discuss augmented assignment statements here?

6.2.1.4 `del` statement

The simple statement

```
del target1, target2, ...
```

recursively deletes each `target` from left to right.

If `target` is a name, then the statement removes the binding of that name from the local or global namespace. If the name is not bound, then the statement raises a `NameError` exception.

If `target` is an attribute reference, subscription, or slicing, the interpreter passes the operation to the primary object involved.

TODO: Expand on what the previous paragraph means. Using special methods?

6.2.2 Compound Statements

A *compound statment* is a construct that contains other statements. This subsection examines three compound statements. We discuss other compound statements later in this textbook.

TODO: Make last sentence above more explicit.

6.2.3 `if` statement

The `if` statement is a conditional statement typical of imperative languages. It is a *compound statement* with the form

```
if cond1:
    statement_list1
elif cond2:    # else with nested if
    statement_list2
elif cond3:
    statement_list3
...
```

```
else:
    statement_listZ
```

where the `elif` and `else` clauses are optional.

When executed, the conditional statement evaluates the `cond` expressions from top to bottom and executes the corresponding `statement_list` for the first condition evaluating to true. If none evaluate to true, then the compound statement executes `statement_listZ`, if it is present.

Note colon terminates each clause header and that the `statement_list` must be indented. Most compound statements are structured in this manner.

6.2.4 `while` statement

The `while` statement is a looping construct with the form

```
while cond:
    statement_list1
else:   # executed after normal exit, not on break
    statement_list2
```

where the `else` clause is optional.

When executed, the `while` repeatedly evaluates the expression `cond`, and, if the condition is true, then the `while` executes `statement_list1`. If the condition is false, then the `while` executes `statement_list2` and exits the loop.

A `break` statement executed in `statement_list1` causes an exit from the loop that does not execute the `else` clause.

A `continue` statement executed in `statement_list1` skips the rest of `statement_list1` and continues with the next condition test.

6.2.5 `for` statement

The `for` statement is a looping construct that iterates over the elements of a sequence. It has the form:

```
for target_list in expression_list:
    statement_list1
else:   # executed after normal exit, not on break
    statement_list2
```

where the `else` clause is optional.

The interpreter:

- evaluates the `expression_list` *once* to get an iterable object (i.e., a sequence)
- creates an iterator to step through the elements of the sequence

- executes `statement_list1` once for each element of the sequence in the order yielded by the iterator

  It assigns each element to the `target_list` (as described above for the assignment statement) before executing `statement_list1`. The variables in the `target_list` may appear as free variables in `statement_list1`.

  The `for` assigns the `target_list` zero or more times as ordinary variables in the local scope. These override any existing values. Any final values are available after the loop.

  If the `expression_list` evaluates to an empty sequence, the the body of the loop is not executed and the `target_list` variables are not changed.

- executes `statement_list2` in the optional `else`, if present, after exhausting the elements of the sequence

The `break` and `continue` statement work as described above for the `while` statement.

It is best to avoid modifying the iteration sequence in the body of the loop. Modification can result in difficult to predict results.

6.3 Function Definitions

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

When the interpreter executes a function *definition*, the interpreter binds the function name in the current local namespace to a *function object*. The function object holds a reference to the current global namespace. The interpreter uses this global namespace when the function is called.

Execution of the function definition does *not* execute the function body. When the function is called, the interpreter then executes the function body.

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

```
def my_func(x, y, z):
    """
    Optional documentation string (docstring)
    """
    statement1
    statement2
    statement3
    return my_loc_var
```

The keyword `def` introduces a function definition. It is followed by the name of the function, a comma-separated parameter list enclosed in parentheses, and a colon.

The body of the functions follows on succeeding lines. The body must be indented from the start of the function header.

Optionally, the first line of the body can be a string literal, called the *documentation string* (or *docstring*). If present, it is stored as the `__doc__` of the function object.

The simple statement

```
return expr1, expr2, ...
```

can only appear within the body of a function definition. When executed during a function call, it evaluates the expressions in its expression list and returns control to the caller of the function, passing back the sequence of values. If the expression list is empty, then it returns the singleton object `None`.

If the last statement executed in a function is not a **`return`**, then the function returns to its caller, returning the value `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 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 `add3` and the following series of commands in the Python REPL.

```
>>> def add3(x, y, z):
...     """Add 3 numbers"""
...     return x + y + z
...
>>> add3(1,2,3)
6
>>> type(add3)
<class `function`>
>>> add3.__doc__
'Add 3 numbers'
>>> x = [add3,1,2,3,6]  # store function object in list
```

```
>>> x
[<function add3 at 0x10bf65ea0>, 1, 2, 3, 6]
>>> x[0](1,2,3)  # retrieve and call function obj
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.

6.4 Class Definitions

A Python *class* is a program construct that defines a new nominal *type* consisting of data attributes and the operations on them.

When the interpreter executes a class *definition*, it binds the class name in the current local namespace to the new *class object* it creates for the class. The interpreter creates a new namespace (for the class's local scope). If the class body contains function or other definitions, these go into the new namespace. If the class contains assignments to local variables, these variables also go into the new namespace.

The class object represents the type. When a program calls a class name 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 interpreter passes the new instance to `__init__`, which initialize 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
```

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 [1,8].

We call a special class object like `type` a *metaclass* because it is a constructor for metaobjects (i.e., class objects) [1,8].

We will look more deeply into these relationships in Chapter 7 when we examine inheritance.

6.5 Module Definitions

A Python *module* is defined in 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 *package* is a directory of Python modules.

A module definition 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 definition as a script from the Python 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.

6.5.1 Using `import`

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

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

def test(x):
    return x
```

We can execute this code in a Python 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`.

6.5.2 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 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`.

6.5.3 Programming conventions

Python 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.

6.5.4 Using `importlib` directly

TODO: Perhaps move the discussion below of the `importlib` a metaprogramming feature, to a later chapter that deals with metaprogramming?

The Python core module `importlib` exposes the functionality underlying the `import` statement to Python 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'>
```

6.6 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 statement, then we can execute the statement dynamically using the `exec` library function as follows:

   ```
   exec(statement)
   ```

   By default, the statement is executed in the current global and local environment, but these environments can be passed in explicitly in optional arguments `globals` and `locals`:

   ```
   exec(statement, globals)
   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.

6.7 Nested Function Definitions

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

Python 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 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 Section 6.9.

  Closure are powerful mechanisms that can be used to implement metaprogramming solutions (e.g., Python's decorators). We discuss those in Chapter 9.

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 and Sussmann [2].)

```
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 sqrt 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 3.7+ source code for the recursive version of `sqrt` is available at this link{type="text/plain} and the looping version at another link.

6.8 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 searches *lexical scope* (also called *static scope*) of a function for variable accesses. (The section on procedural programming paradigm ELIFP [7] Chapter 2 also discusses this concept.)

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 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 defs 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 sqrt of negative number {x}')
```

Note: The Python 3.7+ source code for this version of `sqrt` is available at this link.

6.9 Closures

As discussed in Section 6.7, Python function definitions can be nested inside other functions. Among other capabilities, this enables a Python 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 [9].

```
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 3.7+ source code for `compose2` is available at this link.

6.10 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 }
```

6.11 What Next?

This chapter (6) examined the basic building blocks of Python programs—statements, functions, classes, and modules. Chapter 7 examines classes, objects, and object orientation in more detail.

6.12 Chapter Source Code

TODO

6.13 Exercises

TODO

6.14 Acknowledgements

In Spring 2018, I drafted what is now this chapter as part of the document Basic Features Supporting Metaprogramming, which is Chapter 2 of the 3 chapters of the booklet *Python 3 Reflexive Metaprogramming* [5]. The Spring 2018 material used Python 3.6.

The overall booklet *Python 3 Reflexive Metaprogramming* is inspired by David Beazley's Python 3 Metaprogramming tutorial slides from PyCon'2013 [3]. In particular, I adapted and extended the "Basic Features" material from the terse introductory section of Beazley's tutorial; I attempted to answer questions I had as a person new to Python. Beazley's tutorial draws on material from his and Brian K. Jones' book *Python Cookbook* [4].

In Fall 2018, I divided the Basic Features Supporting Metaprogramming document into 3 chapters—Python Types, Python Program Components (this chapter), and Python Object Orientation. I then revised and expanded each [6]. These 2018 chapers use Python 3.7.

This chapter seeks to be compatible with the concepts, terminology, and approach of my textbook *Exploring Languages with Interpreters and Functional Programming* [7], in particular of Chapters 2, 3, 5, 6, 7, 11, and 21.

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating (e.g., using CSS), constructing a unified bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

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

6.15 Terms and Concepts

TODO

6.16 References

[1] 1991. *The art of the metaobject protocol*. MIT Press, Cambridge, Massachusetts, USA.

[2] Harold Abelson and Gerald Jay Sussman. 1996. *Structure and interpretation of computer programs (SICP)* (Second ed.). MIT Press, Cambridge, Massachusetts, USA. Retrieved from https://mitpress.mit.edu/sicp/

[3] David Beazley. 2013. Python 3 metaprogramming (tutorial). Retrieved from http://www.dabeaz.com/py3meta/

[4] David Beazley and Brian K. Jones. 2013. *Python cookbook* (Third ed.). O'Reilly Media, Sebastopol, California, USA.

[5] H. Conrad Cunningham. 2018. Python 3 reflexive metaprogramming. University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/csci658/notes/PythonMetaprogramming/Py3RefMeta.html

[6] H. Conrad Cunningham. 2018. Multiparadigm programming with Python 3. University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/csci556/Py3MPP/Ch05/05_Python_Types.html

[7] H. Conrad Cunningham. 2022. *Exploring programming languages with interpreters and functional programming (ELIFP)*. University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/ELIFP/ELIFP.pdf

[8] Ira R. Forman and Scott Danforth. 1999. *Putting metaclasses to work: A new dimension in object-oriented programming*. Addison-Wesley, Boston Massachusetts, USA.

[9] Mathieu Larose. 2013. Function composition in python (blog post). Retrieved from https://mathieularose.com/function-composition-in-python

[10] Python Software Foundation. 2022. Python 3 documentation. Retrieved from https://docs.python.org/3/