Multiparadigm Programming with Python (PyMPP)

H. Conrad Cunningham

10 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.

5 Python Types

5.1 Chapter Introduction

The goals of this chapter (5) are to:

- examine the general concepts of type systems
- explore Python's type system and built-in types

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

5.2 Type System Concepts

The term *type* tends to be used in many different ways in programming languages. What is a type?

Chapter 7) on object-based paradigms discusses the concept of type in the context of object-oriented languages. This chapter first examines the concept more generally and then examines Python's builtin types.

5.2.1 Types and subtypes

Conceptually, a *type* is a set of values (i.e., possible states or objects) 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` an 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* [24,39].

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.

5.2.2 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 in a particular context and at a particular time during execution. The type may be constrained by a declaration of the variable.

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

5.2.3 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, JavaScript, 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.

5.2.4 Nominal and structural

In a language with *nominal typing*, the type of value is based on the type *name* assigned when the value is created. Two values 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 a value is based on the *structure* of the value. Two values 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 an example of a primarily structurally typed language.

5.2.5 Polymorphic operations

Polymorphism refers to the property of having "many shapes". In programming languages, we are primarily interested in how *polymorphic* function names (or operator symbols) are associated with implementations of the functions (or operations).

In general, two primary kinds of polymorphic operations exist in programming languages:

1. *Ad hoc polymorphism*, in which the same function name (or operator symbol) can denote different implementations depending upon how it is used in an expression. That is, the implementation invoked depends upon the types of function's arguments and return value.

   There are two subkinds of ad hoc polymorphism.

   a. *Overloading* refers to ad hoc polymorphism in which the language's compiler or interpreter determines the appropriate implementation to invoke using information from the context. In statically typed languages, overloaded names and symbols can usually be bound to the intended implementation at *compile time* based on the declared types of the entities. They exhibit *early binding*.

      Consider the language Java. It overloads a few operator symbols, such as using the `+` symbol for both addition of numbers and concatenation of strings. Java also overloads calls of functions defined with the same name but different signatures (patterns of parameter types and return value). Java does not support user-defined operator overloading; C++ does.

      Haskell's *type class* mechanism implements overloading polymorphism in Haskell. There are similar mechanisms in other languages such as Scala and Rust.

   b. *Subtyping* (also known as *subtype polymorphism* or *inclusion polymorphism*) refers to ad hoc polymorphism in which the appropriate implementation is determined by searching a hierarchy of types. The function may be defined in a supertype and redefined (overridden) in subtypes. Beginning with the actual types of the data involved, the program searches up the type hierarchy to find the appropriate implementation to invoke. This usually occurs at runtime, so this exhibits *late binding*.

The object-oriented programming community often refers to inheritance-based subtype polymorphism as simply *polymorphism*. This is the polymorphism associated with the class structure in Java.

Haskell does not support subtyping. Its type classes do support *class extension*, which enables one class to inherit the properties of another. However, Haskell's classes are not types.

2. *Parametric polymorphism*, in which the same implementation can be used for many different types. In most cases, the function (or class) implementation is stated in terms of one or more type parameters. In statically typed languages, this binding can usually be done at compile time (i.e., exhibiting early binding).

   The object-oriented programming (e.g., Java) community often calls this type of polymorphism *generics* or *generic programming*.

   The functional programming (e.g., Haskell) community often calls this simply *polymorphism*.

5.2.6 Polymorphic variables

A *polymorphic variable* is a variable that can "hold" values of different types during program execution.

For example, a variable in a dynamically typed language (e.g., Python) is polymorphic. It can potentially "hold" any value. The variable takes on the type of whatever value it "holds" at a particular point during execution.

Also, a variable in a nominally and statically typed, object-oriented language (e.g., Java) is polymorphic. It can "hold" a value its declared type or of any of the subtypes of that type. The variable is declared with a static type; its value has a dynamic type.

A variable that is a parameter of a (parametrically) polymorphic function is polymorphic. It may be bound to different types on different calls of the function.

5.3 Python Type System

What about Python's type system?

5.3.1 Objects

Python is *object-based* [13, Ch. 3]; it treats all data as *objects*.

A Python object has the following *essential* characteristics of objects [13, Ch. 3]:

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 pbject has one of the two *important but nonessential* characteristics of objects [13, Ch. 3]. Python does:

d. *not* enforce *encapsulation* of the state within the object, instead relying upon programming conventions and name *obfuscation* 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 Chapter 7, each object has a distinct dictionary, the directory, that maps the local names to the data attributes and operations.

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

- `obj.data` accesses the attribute `data` of `obj`
- `obj.op` accesses operation `op` of `obj`
- `obj.op()` invokes operation `op` of `obj`, passing any arguments in a comma-separated list between the parentheses

Some objects are immutable and others are mutable. The states (i.e., values) of:

- *immutable* objects (e.g., numbers, booleans, strings, and tuples) cannot be changed after creation
- *mutable* objects (e.g., lists, dictionaries, and sets) can be changed in place after creation

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

All Python objects have a type.

5.3.2 Types

In terms of the discussion in Section {#sec:type-system-concepts}, all Python 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 variable is bound to an object by an assignment statement or its equivalent. Python variables are thus *dynamically typed*, as are Python expressions.

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

5.4 Built-in Types

Python 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 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 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
- Set types
- Other types (e.g., callable, class, module, and user-defined object types)

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

5.4.1 Singleton types

Python has single-element types used for special purposes. These include `None` and `NotImplemented`.

5.4.1.1 `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. For example, this is the value returned by Python procedures.

5.4.1.2 `NotImplemented`

The name `NotImplemented` also denotes a value of a singleton type. Python programs normally use this value to mean that an arithmetic or comparison operation is not implemented on the operand objects.

5.4.2 Number types

Core Python supports four types of numbers:

- integers
- real numbers
- complex numbers
- Booleans

5.4.2.1 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'>
>>> type(99999999999999999999999999999999)
<class 'int'>
```

5.4.2.2 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
```

5.4.2.3 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
```

5.4.2.4 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 modern Python programs.

5.4.2.5 Truthy and falsy values

Python 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 on the object, returns `False`
- a special method `__len__()` that returns `0`

Note: We discuss special methods Chapter 7.

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.

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

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

5.4.3.1.1 `str` Type `str` (string) denotes sequences of text characters—that is, of *Unicode code points* in Python. 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'>
```

5.4.3.1.2 `tuple` Type `tuple` type 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 themselves be mutable.

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

5.4.3.1.3 `range` The `range` type denotes an immutable sequence of numbers. It is commonly used to specify loop controls.

- `range(stop)` denotes the sequence of integers that starts from 0, increases by steps of 1, and stops at `stop-1`; if `stop <= 0`, the range is empty.
- `range(start, stop)` denotes the sequence of integers that starts from `start`, increases by steps of 1, and stops at `stop-1`; if `stop <= start`, the range is empty.
- `range(start, stop, step)` denotes the sequence of integers that starts from `start` with a *nonzero* stepsize of `step`.

  If `step` is positive, the sequence increases toward `stop-1`; if `stop <= start`, the range is empty.

  if negative, the sequence decreases toward `stop+1`.

  A range is a lazy data structure. It only yields a value if the output is needed.

```
>>> list(range(5))
[0, 1, 2, 3, 4]
>>> list(range(1, 5))
[1, 2, 3, 4]
>>> list(range(0, 9, 3))
[0, 3, 6]
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
>>> list(range(5, 0, -1))
[5, 4, 3, 2, 1]
>>> list(range(0))
[]
>>> list(range(1, 0))
[]
>>> list(range(0, 0))
[]
```

5.4.3.1.4 `bytes` Type `bytes` type 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>
```

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

5.4.3.2.1 `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]
```

5.4.3.2.2 `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>
```

5.4.4 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 can be any *hashable* object. That is, the key can be any immutable object or an object that always gives the same hash value. 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'}
```

5.4.5 Set Types

A *set* is an *unordered collection* of distinct *hashable* objects.

There are two built-in set types—`set` and `frozenset`.

5.4.5.1 `set`

Type `set` denotes a mutable collection.

We can create a nonempty set by putting a comma-separated list of elements between braces as well as by using the `set` constructor.

For example, sets `sx` and `sy` below have the same elements. The operation `|=` adds the elements of the right operand to the left.

```
    >>> sx = { 'Dijkstra', 'Hoare', 'Knuth' }
    >>> sx
    {'Knuth', 'Hoare', 'Dijkstra'}
    >>> sy = set(['Knuth', 'Dijkstra', 'Hoare'])
    >>> sy
<   {'Knuth', 'Hoare', 'Dijkstra'}
    >>> sx == sy
    True
    >>> sx.add('Turing')  # add element to mutable set
    >>> sx
    {'Turing', 'Knuth', 'Hoare', 'Dijkstra'}
```

5.4.5.2 `frozenset`

Type `frozenset` denotes an immutable collection.

We can extend the `set` example above as follows:

```
    >>> fx = frozenset(['Dijkstra', 'Hoare', 'Knuth'])
    >>> fx
    frozenset({'Knuth', 'Hoare', 'Dijkstra'})
    >>> fy = frozenset(sy)
    >>> fy
```

```
frozenset({'Knuth', 'Hoare', 'Dijkstra'})
>>> fx.add('Turing')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
```

5.4.6 Other object types

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

TODO: Perhaps be more specific about later chapters.

5.5 What Next?

TODO

5.6 Chapter Source Code

TODO, if needed.

5.7 Exercises

TODO

5.8 Acknowledgements

In Spring 2018, I wrote the general Type System Concepts section as a part of a chapter that discusses the type system of Python 3 [9] to support my use of Python in graduate CSci 658 (Software Language Engineering) course.

In Summer 2018, I revised the section to become Section 5.2 in Chapter 5 of the evolving textbook *Exploring Languages with Interpreters and Functional Programming* (ELIFP) [13]. I also moved the "Kinds of Polymorphism" discussion from the 2017 List Programming chapter to the new subsection "Polymorphic Operations". This textbook draft supported my Haskell-based offering of the core course CSci 450 (Organization of Programming Languages).

In Fall 2018, I copied the general concepts section from ELIFP and recombined it with the Python-specific content [9] to form this chapter to support my Python-based offering of the elective course CSci 556 (Multiparadigm Programming) and for a posible future book [10].

In Spring 2019, I extracted the general concepts discussion [10] to create a chapter [11] for use in my Scala-based offering of CSci 555 (Functional Programming).

The type concepts discussion draws ideas from various sources:

- my general study of a variety of programming, programming language, and software engineering over three decades [5–7,17–19,21,24–28,33,34,37].

- the *Wikipedia* articles on the Liskov Substitution Principle [39], Polymorphism [40], Ad Hoc Polymorphism [42], Parametric Polymorphism [43], Subtyping [44], and Function Overloading [41]

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 additional separately developed materials, reformatting the document (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.

5.9 Terms and Concepts

TODO: revise for the current content

Object, object characteristics (state, operations, identity, encapsulation, independent lifecycle), immutable vs. mutable, type, subtype, Liskov Substitution Principle, types of constants, variables, and expressions, static vs. dynamic types, declared and inferred types, nominal vs. structural types, polymorphic operations (ad hoc, overloading, subtyping, parametric/generic), early vs. late binding, compile time vs. runtime, polymorphic variables, duck typing, protocol, interface, REPL, singleton types (`None` and `NotImplemented`), number types (`int`, `float`, `complex`, `bool`, `False`, falsy, `True`, truthy), immutable sequence types (`str`, `tuple`, `range`, `bytes`), mutable sequence types (`list`, `bytearray`), mapping types (`dict`, key and value), set types (`set`, `frozenset`),other types.

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 [30] 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 [1,14].

We call a special class object like `type` a *metaclass* because it is a constructor for metaobjects (i.e., class objects) [1,14].

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].)

```python
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 [13] 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 [23].

```
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* [9]. 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 [10]. 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* [13], 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

7 Python Object Orientation

7.1 Chapter Introduction

Chapter 6 examined the basic building blocks of Python programs—statements, functions, classes, and modules.

This chapter (7) examines classes, objects, and object orientation in more detail.

TODO: Chapter goals.

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

7.2 Class and Instance Attributes

As we saw in Chapter 6, 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 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 [35]:

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 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.

```python
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 discuss decorators in Chapter 9.

Now consider a Python 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 obj, store in var 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`.

The Python 3.7+ source code for the class `Dummy` is available in file `dummy1,py`.

7.3 Object Dictionaries

As we noted in Chapter 5, each Python 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 programs can access this dictionary directly.

Again consider the `Dummy` class we examined in Section 7.2. Let's look at dictionary for this class and an instance in the Python 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?

7.4 Special Methods and Operator Overloading

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

Python 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 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 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
```

The Python 3.7+ source code for the class `SparseArray` is available in file `sparse_arrat1,py`.

7.5 Object Orientation

TODO: Remove any unnecessary duplication among this discussion and similar discussions in Chapters 3 and 5.

Chapter 3, Object-Based Paradigms, of *Exploring Languages with Interpreters and Functional Programming* (ELIFP) [13] discusses 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 polymorph

We discuss Python's objects in Chapter 5 and classes in Chapter 6.

Now let's consider the other two components of the general object model in relation to Python.

7.5.1 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* [24,39].

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, 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. 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`.

```python
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 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 3.7+ source code for the above version of the `P` class hierarchy is available in file `inherit1.py`.

7.5.1.1 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 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 Chapter 6, we call class objects *metaobjects*; they are constructors for ordinary objects [1,14].

Also as we noted in Chapter 6, we call special class objects like `type` *metaclasses*; they are constructors for metaobjects (i.e., class objects) [1,14].

Note that classes `object` and `type` have special – almost “magical” – relationships with one another [31:593–595].

- 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 7.1 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.

7.5.1.2 Replacement and refinement There are two general approaches for overriding methods in subclasses [8]:

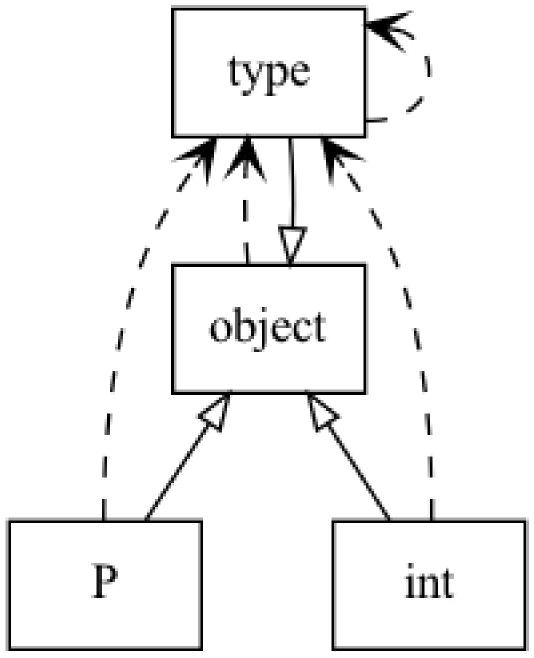

Figure 7.1: Python Class Model

- *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.

- *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 class hierarchy example above, the expression `super().process()` in subclass `C` calls the `process` method of its superclass `P`.

7.5.2 Subtype polymorphism

The concept of *polymorphism* (literally "many forms") means the ability to hide different implementations behind a common interface. As we saw in Chapter 5, polymorphism appears in several forms in programming languages. Here we examine one form.

In the Python 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 than in Java.

In Python, 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 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'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 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'
```

7.5.3 Multiple Inheritance

TODO: Discuss multiple inheritance. Issues include the diamond problem, Python syntax and semantics, and method resolution order.

7.6 What Next?

TODO

7.7 Chapter Source Code

TODO

7.8 Exercises

TODO

7.9 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* [9]. 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, and Python Object Orientation (this chapter). I then revised and expanded each [10]. 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* [13], 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.

7.10 Terms and Concepts

TODO

8 Metaprogramming

8.1 Chapter Introduction

TODO: Introduction and other missing parts. These are updated consistently for use with other chapters.

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

8.2 Definition

Basically, *metaprogramming* is writing code that writes code [45,46].

Metaprogramming: the writing of computer programs that can treat computer programs as their data. A program can read, generate, analyze, and/or transform other programs, or even modify itself while running)

We often do metaprogramming in our routine programming tasks but do not call it that.

- Our web applications may generate HTML, JavaScript, and CSS code to enable development of a browser-based user interface.
- Our Java programs may use `instanceof` to check the type of objects or otherwise manipulate itself with the Java reflection package.
- Our C programs may use macros to define new features in terms of existing features [36].
- The PIC little language processor takes a program expressed in an external textual language that describes a picture and generates output expressed in another language that gives instructions to a display program [22,32].

Under the above definition, much of our study of domain-specific languages uses metaprogramming.

8.3 Reflexive Metaprogramming

TODO: Decide how to consistently refer to my DSL course examples. Should be consistent with the separate longer Domain-Specific Languages document.

The internal Survey DSL and Lair Configuration DSL are examples of *reflexive (or reflective) metaprogramming* [45,46].

Reflexive metaprogramming: the writing of computer programs that manipulate themselves as data.

This manipulation may be at compile time, involving a phase of transformations in the code before the final program is generated. Or it may be at runtime, involving manipulation of the program's metamodel or generation of new code that is dynamically executed within the program.

The Survey DSL is a Ruby internal DSL. It takes advantage of Ruby's metaprogramming facilities such as the abilities to trap calls to undefined methods, add methods or variables dynamically to existing objects at runtime, and execute dynamically generated strings as Ruby code. It also uses Ruby's first-class functions (closures) and flexible syntax – although these are not technically metaprogramming features of Ruby.

The Lair Configuration DSL programs use the metaprogramming features of Lua and Python in similar ways.

Consider relatively common languages and their metaprogramming features.

- Java is a statically typed, compiled language. What are metaprogramming features available in Java?

  It has dynamic class loaders, a reflection API, annotation processing, dynamic method invocation (a JVM feature), JVM bytecode manipulation (mostly with external tools), etc. Java 8+ also has first-class functions and other features useful in metaprogramming.

- Lua is a dynamically typed, interpreted language. What are the metaprogramming features available in Lua?

  It has metatables, metamethods, manipulation of environments, a debug library (introspection/reflection features), `loadfile` and `loadstring` functions to dynamically execute code, extensions in C, etc.

What about Python?

The reflexive metaprogramming features of Python 3.7 and beyond is the primary topic of this set of lecture notes.

8.4 Why Study Reflexive Metaprogramming?

In everyday application programming, we often use the products developed by metaprogrammers, but we seldom use the techniques directly.

In everyday programming, use of reflexive metaprogramming techniques should not be one of our first approaches to a problem. We first should explore techniques supported by core language, its standard libraries, and stable extension packages.

If no acceptable solution can be found, then we can consider solutions that use reflexive metaprogramming techniques. We should approach metaprogramming with great care because these techniques can make programs difficult to understand and can introduce vulnerabilities into our programs. We should design, implement, test, and document the programs rigorously.

However, reflexive metaprogramming can be an important tool in a master programmer's toolbox. If our jobs are to develop software frameworks, libraries, APIs, or domain-specific languages, we can use these techniques and features to develop powerful products that hide the complexity from the application programmer.

Even when our jobs are primarily application programming, understanding reflexive metaprogramming techniques can improve our abilities to use software frameworks, libraries, and APIs effectively.

8.5 Reflexive Metaprogramming in Python

TODO: Update this to better reflect, what the final notes cover and include forward references as appropriate.

The reflexive metaprogramming features of Python include:

1. Decorators
2. Metaclasses
3. Descriptors
4. Import hooks
5. Context managers
6. Annotations (e.g., type hints)
7. Abstract Syntax Tree (AST) manipulation
8. Frame hacks
9. Execution of strings as Python 3 code (`exec`, `eval`)
10. *Monkeypatching* (i.e., direct dynamic manipulation of attributes and methods at runtime)

We have already used the final two in our implementation of domain-specific languages. We will look at some of the others in these notes. In particular, Chapter 9 looks at use of decorators and metaclasses.

8.6 What Next?

TODO

8.7 Chapter Source Code

TODO

8.8 Exercises

TODO: Decide if any are appropriate.

8.9 Acknowledgements

I developed these notes in Spring 2018 for use in CSci 658 Software Language Engineering [9]. The Spring 2018 version used Python 3.6.

Teaching a special topics course on "Ruby and Software Development" in Fall 2006 kindled my interests in domain-specific languages and metaprogramming.

Building on these interests, I taught another special topics course on "Software Language Engineering" in which I focused on Martin Fowler's work on domain-specific languages [15]; I subsequently formalized this as CSci 658. (I have collected some overall ideas on in my notes on Domain Specific Languages [12].)

The overall set of notes on Python 3 Reflexive Metaprogramming is inspired by David Beazley's Python 3 Metaprogramming tutorial from PyCon'2013 [3]. In particular, some chapters adapt Beazley's examples. Beazley's tutorial draws on material from his and Brian K. Jones' book *Python Cookbook* [4].

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 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.

8.10 Terms and Concepts

TODO: Update

Metaprogramming, reflexive metaprogramming.

9 Python Decorators and Metaclasses

In this chapter, we look at metaprogramming using Python decorators and metaclasses. To do so, we consider a simple tracing debugger case study, adapted from David Beazley's `debugly` example from his metaprogramming tutorial [3].

TODO: Chapter goals.

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

9.1 Basic Function-Level Debugging

9.1.1 Motivating example

Suppose we have a Python function `add`:

```
def add(x, y):
    'Add x and y'
    return x + y
```

A simple way we can approach debugging is to insert a `print` statement into the function to trace execution, as follows:

```
def add(x, y):
    'Add x and y'
    print('add')
    return x + y
```

However, suppose we need to debug several similar functions simultaneously. Following the above approach, we might have code similar to that in the example below.

```
def add(x, y):
    'Add x and y'
    print('add')
    return x + y

def sub(x, y):
    'Subtract y from x'
    print('sub')
    return x - y

def mul(x, y):
    'Multiply x and y'
    print('mul')
    return x * y

def div(x, y):
    'Divide x by y'
```

```
        print('div')
        return x / y
```

We insert basically the same code into every function.

This code is unpleasant because it violates the Abstraction Principle.

9.1.2 Abstraction Principle, staying DRY

The *Abstraction Principle* states, "Each significant piece of functionality in a program should be implemented in just one place in the source code." [29:339]. If similar functionality is needed in several places, then the common parts of the functionality should be separated from the variable parts.

The common parts become a new programming abstraction (e.g., a function, class, abstract data type, design pattern, etc.) and the variable parts become different ways in which the abstraction can be customized (e.g.„ its parameters).

The approach encourages reuse of both design and code. Perhaps more importantly, it can make it easier to keep the similar parts consistent as the program evolves.

Andy Hunt and Dave Thomas [20:26–33] articulate a more general software development principle *Don't Repeat Yourself*, known by the acronym *DRY*.

In an interview [38], Thomas states, "DRY says that every piece of system knowledge should have one authoritative, unambiguous representation. … A system's knowledge is far broader than just its code. It refers to database schemas, test plans, the build system, even documentation."

Our goal is to keep our Python code DRY, not let it get WET ("Write Everything Twice" or "Wasting Everyone's Time" or "We Enjoy Typing" [47].)

9.1.3 Function decorators

To introduce an appropriate abstraction into the previous set of functions, we can use a Python function decorator.

A *function decorator* is a higher-order function that takes a function as its argument, wraps another function around the argument, and returns the wrapper function.

The wrapper function has the same parameters and same return value as the function it wraps, except it does extra processing when it is called. That is, it "decorates" the original function.

TODO: Show Wikipedia citation because of link.

Remember that Python functions are objects. Python's decorator function concept is thus a special case of the Decorator design pattern, one of the classic Gang of Four patterns for object-oriented programming [16]. The idea of this pattern is to wrap one object with another object, the decorator, that has the same interface but enhanced behavior. The decoration is usually done at runtime even in a statically typed language like Java.

TODO: Perhaps expand on the Decorator design pattern and give a diagram.

9.1.4 Constructing a debug decorator

In the motivating example above, we want to decorate a function like `add(x,y)` by wrapping it with another function that prints the function name `add` before doing the addition operation. The wrapped function can then take the place of the original `add` in the program.

Let's construct an appropriate decorator in steps.

In general, suppose we want to decorate a function named `func` that takes some number of positional and/or keyword arguments. That is, the function has the general signature:

```
func(*args, **kwargs)
```

Note: For more information on the above function calling syntax, see the discussion on Function Calling Conventions in Chapter 6.

In addition, suppose we want to print the content of the variable `msg` before we execute `func`.

As our first step, we define function `wrapper` as follows:

```
def wrapper(*args, **kwargs):
    print(msg)
    return func(*args, **kwargs)
```

As our second step, we define a decorator function `debug` that takes a function `func` as its argument, sets local variable `msg` to `func`'s name, and then creates and returns the function `wrapper`.

Function `debug` can retrieve the function name by accessing the `__qualname__` attribute of the `func` object. Attribute `__qualname__` holds the fully qualified name.

```
def debug(func):
    msg = func.__qualname__
    def wrapper(*args, **kwargs):
        print(msg)
        return func(*args, **kwargs)
    return wrapper
```

Function `debug` returns a closure that consists of the function `wrapper` plus the the local environment in which `wrapper` is defined. The local environment includes the argument `func` and the variable `msg` and their values.

Note: For more information about the concepts and techniques used above, see the discussion of Nested Function Definitions, Lexical Scope, and Closures in Chapter 6.

It seems sufficient to assign the closure returned by `debug` to the name of `func` as shown below for `add`.

```
def add(x, y):
    'Add x and y' # docstring (documentation)
    return x + y
add = debug(add)
```

But this does not work as expected as shown in the following REPL session.

```
>>> add(2,5)
add
7
>>> add.__qualname__
debug.<locals>.wrapper
>>> add.__doc__
None
```

The closure returned by `debug` computes the correct result. However, it does not have the correct metadata, as illustrated above by the display of the name (`__qualname__`) and the docstring (`__doc__`) metadata.

To make the use of the decorator `debug` transparent to the user, we can apply the function decorator `@wraps` defined in the standard module `functools` as follows.

```
def debug(func):
    msg = func.__qualname__
    @wraps(func)
    def wrapper(*args, **kwargs):
        print(msg)
        return func(*args, **kwargs)
    return wrapper

def add(x, y):
    'Add x and y' # docstring (documentation)
    return x + y
add = debug(add)
```

The `@wraps(func)` decorator call above sets function `wrapper`'s metadata — it's attributes `__module__`, `__name__`, `__qualname__`, `__annotations__`, and `__doc__` — to the same values as `func`'s metadata.

With this new version of the `debug` decorator, the decoration of `add` now works transparently.

```
>>> add(2,5)
add
7
>>> add.__qualname__    add
>>> add.__doc__
Add x and y
```

Finally, because the definition of a function like `add` and the application of the `debug` decorator function usually occur together, we can use the decorator *syntactic sugar* `@debug` to conveniently designate the definition of a decorated function. The `debug` decorator function can be defined in a separate module.

```
@debug
def add(x, y):
    'Add x and y'
    return x + y
```

9.1.5 Using the debug decorator

Given the `debug` decorator as defined in the previous subsection, we can now simplify the motivating example.

We decorate each function with `@debug` but give no other details of the implementation here. The debug facility is implemented in one place but used in many places. The implementation supports the DRY principle.

```
@debug
def add(x, y):
    'Add x and y'
    return x + y

@debug
def sub(x, y):
    'Subtract y from x'
    return x - y

@debug
def mul(x, y):
    'Multiply x and y'
    return x * y

@debug
def div(x, y):
    'Divide x by y'
    return x / y
```

Note: The Python 3.7+ source code for the above version of `debug` is available in linked file `debug4.py`.

9.1.6 Case study review

So far in this case study, we have implemented a simple debugging facility that:

- is implemented once in a place separate from its use
- is thus easy to modify or disable totally
- can be used without knowing its implementation details

9.1.7 Variations

Now let's consider a couple of variations of the debugging decorator implementation.

9.1.7.1 Logging

One variation would be to use the Python logging module to log the messages instead of just printing them [3].

The details of logging are not important here, but note that we only need to make three changes to the `debug` implementation. We do not need to change the user code.

```
from functools import wraps
import logging  # (1) logging module

def debug(func):
    # (2) get the Logger for func's module
    log = logging.getLogger(func.__module__)
    msg = func.__qualname__
    @wraps(func)
    def wrapper(*args, **kwargs):
        log.debug(msg)  # (3) log msg
        return func(*args, **kwargs)
    return wrapper
```

Note: The Python 3.7+ source code for the above version of `debug` is available in linked file `debuglog1.py`.

9.1.7.2 Optional disable

Another variation of the debugging decorator would be to only enable debugging when a particular environment variable is set [3]. In this variation, we only need to make two changes to the `debug` implementation.

```
from functools import wraps
import os  # (1) import os interface

def debug(func):
    # (2) debug only if environment variable set
    if 'DEBUG' not in os.environ:
        return func
    msg = func.__qualname__
    @wraps(func)
    def wrapper(*args, **kwargs):
        print(msg)
        return func(*args, **kwargs)
    return wrapper
```

Note: The Python 3.7+ source code for the above version of `debug` is available in linked file `debugopt1.py`.

9.2 Extended Function-Level Debugging

Now we can extend the capability of our simple tracing debugger [3].

9.2.1 Motivating example

Suppose, for whatever reason, we want to add a prefix string to the debugging message that may differ from one use of `@debug` to another. Again consider the set of arithmetic functions.

```
    def add(x, y):
        'Add x and y'
        print('***add')
        return x + y

    def sub(x, y):
        'Subtract y from x'
        print('@@@sub')
        return x - y

    def mul(x, y):
        'Multiply x and y'
        print('***sub')
        return x * y

    def div(x, y):
        'Divide x by y'
        print('div')
        return x / y
```

We implement the needed capability by using function decorators with arguments.

9.2.2 Decorators with arguments

We can construct decorators that take arguments other than the function to be decorated.

Consider the following use of decorator `deco`:

```
@deco(args)
def func():
    # some body code
```

The above translates into the following decorator call and assignment:

```
func = deco(args)(func)
```

The right-hand side denotes the chaining of two function calls. The system first calls function `deco` passing it the first argument list `(args)`. This call returns a function, which is in turn called with the second argument list, variable `(func)`.

The outer function call establishes a local environment in which the variables in `args` are defined. In this environment, we define a normal decorator as we did before.

9.2.3 Prefix decorator

We can thus define the outer layer of a prefix decorator with a function with parameter `prefix` that defaults to the empty string.

```
def debug(prefix=''):
    def deco(func):
        # normal debug decorator body
    return deco
```

The full definition of the prefix decorator is shown below. If no argument is given to `debug`, the behavior is (almost) the same as the previous `debug` decorator function.

```
from functools import wraps

def debug(prefix=''):
    def deco(func):
        msg = prefix + func.__qualname__
        @wraps(func)
        def wrapper(*args, **kwargs):
            print(msg)
            return func(*args, **kwargs)
        return wrapper
    return deco
```

In this formulation, `prefix` can be given as either a positional or keyword argument.

We can apply the new prefix debug decorator to our motivating example functions as follows. Note that the `prefix` strings vary among the different occurrences.

```
@debug(prefix='***')
def add(x,y):
    'Add x and y'
    return x+y

@debug(prefix='@@@')
def sub(x, y):
    'Subtract y from x'
    return x - y

@debug('***')
def mul(x, y):
    'Multiply x and y'
    return x * y

@debug()  # parentheses needed!
def div(x, y):
    'Divide x by y'
    return x / y
```

Note: The Python 3.7+ source code for the above version of `debugprefix` is available in linked file `debugprefix1.py`.

9.2.4 Reformulated prefix decorator

By a clever use of default arguments and partial application of a function to its arguments, we can transform the definition of the prefix decorator above to one that does not involve a nested definition.

```
from functools import wraps, partial

def debug(func = None, *, prefix = ''):
    if func is None:
        return partial(debug, prefix=prefix)
    msg = prefix + func.__qualname__
    @wraps(func)
    def wrapper(*args, **kwargs):
        print(msg)
        return func(*args, **kwargs)
    return wrapper
```

If we call the `debug` decorator function with the single keyword argument `prefix`, then the `func` argument defaults to `None`. In this case, the `if` statement causes `debug` to call itself with that `prefix` argument and the decorated function (that follows the `@debug` annotation in the user-level code or occurs in a second argument list) as the `func` argument.

Note: The `functools.partial` function takes a function (object) and a group of positional and/or keyword arguments, *partially applies* the function to those arguments, then returns the resulting function (object). The returned function behaves like the original function except that it has the argument values supplied to `partial` as its default parameter values.

If we call the `debug` decorator function with no keyword arguments, then parameter `prefix` defaults to the empty string and `func` is the decorated function (e.g., that follows the `@debug` annotation).

If we call `debug` with both `func` and `prefix` arguments, then it works as we expect. This case is not used with the `@debug` annotation.

```
@debug(prefix='***')
def add(x,y):
    'Add x and y'
    return x+y

@debug(prefix='@@@')
def sub(x, y):
    'Subtract y from x'
    return x - y

@debug(prefix='***')
def mul(x, y):
    'Multiply x and y'
    return x * y

@debug  # no parentheses required, but okay if given
def div(x, y):
    'Divide x by y'
    return x / y
```

Unlike the previous formulation of the prefix decorator, the `prefix` string must be supplied as a prefix argument.

Note: The Python 3.7+ source code for the above version of `debugprefix` is available in linked file `debugprefix2.py`.

9.3 Class-Level Debugging

9.3.1 Motivating example

Consider the class `Account` below for a simple bank account.

Suppose we want to debug all the methods using the simple debugging package we developed above.

```
class Account:
    def __init__(self):
        self._bal = 0

    @debug
    def deposit(self,amt):
        self._bal += amt

    @debug
    def withdraw(self,amt):
        if amt <= self._bal:
            self._bal -= amt
        else:
            print(f'Insufficient funds for withdrawal of {amt}')

    @debug
    def get_balance(self):
        return self._bal

    def __str__(self):
        return f'Account with balance {self._bal}'
```

Note: The Python 3.7+ source code for the above version of `Account` is available in linked file `account2.py`.

9.3.2 Class-level debugger

The `Account` example above is repetitive (not DRY). Can we do the decoration all at once?

Yes, we can define a class decorator `debugmethods` as shown below (where `debug` is the function-level prefix decorator defined above). A *class decorator* is a higher-order function that takes a class as its argument, modifies the class in some way, and then returns the modified class.

```
    def debugmethods(cls):                          # Notes
        for name, val in vars(cls).items():     # (1) (2) (3)
            if callable(val):                       # (4)
                setattr(cls, name, debug(val)) # (5) (6)
        return cls                                  # (7)
```

The idea here is that the program walks through the class dictionary, identifies callable objects (e.g., methods), and wraps each with a function decorator.

Consider the numbered comments in the above code.

1. The built-in function call `vars(cls)` returns the dictionary (i.e., `__dict__`) associated with the (class) object `cls`.

2. The dictionary method call `items()` returns the list of key-value pairs in the dictionary.

3. The "`for name, val in`" statement loops through the pairs in the list, successively binding each key to `name` and value to `val`.

4. The built-in function call `callable(val)` returns `True` if `val` appears callable, `False` if not. (These are likely instance methods.)
5. The call `debug(val)` applies the function-level prefix debugger we defined above to the method `val`. That is, it wraps the method with function decorator `debug`.
6. The built-in function call `setattr(cls, name, debug(val))` sets the `name` attribute of object `cls` (i.e., in its dictionary) to the value `debug(val)`.
7. The decorator function `debugmethods` returns the modified class object `cls` in place of the original class.

The code below shows the application of this new decorator to the `Account` class.

```
@debugmethods
class Account:
    def __init__(self):
        self._bal = 0

    def deposit(self,amt):
        self._bal += amt

    def withdraw(self,amt):
        if amt <= self._bal:
            self._bal -= amt
        else:
            print(f'Insufficient funds for withdrawal of {amt}')

    def get_balance(self):
        return self._bal

    def __str__(self):
        return f'Account with balance {self._bal}'
```

Note: The Python 3.7+ source code for the above version of `Account` is available in linked file `account3.py`.

A single decorator application handles all the method definitions within the class.

Well, not quite!

It does not decorate class or static methods, such as the following which can be added to class `Account`.

```
class Account:
...
    @classmethod
    def classname(cls):
        return cls.__name__

    @staticmethod
    def warn(msg):
        print(f'Warning: {msg}')
```

Note: The Python 3.7+ source code for the extended version of `Account` is available in linked file `account4.py`.

TODO: Explain why this does not work.

9.3.3 Variation: Attribute access debugging

Suppose instead of printing a message on every call of a method, we do so for each access to an attribute.

We can do this by rewriting part of the class as shown below. In particular, we give a new implementation for the special method `__getattribute__`.

```
def debugattr(cls):
    orig_getattribute = cls.__getattribute__

    def __getattribute__(self, name):
        print(f'Get: {name}')
        return orig_getattribute(self, name)

    cls.__getattribute__ = __getattribute__
    return cls
```

The special method `__getattribute__` is called to implement accesses to "regular" attributes of the class. It is not called on accesses to other special methods such as `__init__` and `__str__`.

In the above, we save the original implementation of the method and then call it to complete the access once we have printed an appropriate debugging message.

In the example below, we decorate the `Account` class with `@debugattr`.

```
@debugattr
class Account:
    def __init__(self):
        self._bal = 0

    def deposit(self,amt):
        self._bal += amt

    def withdraw(self,amt):
        if amt <= self._bal:
            self._bal -= amt
        else:
            print(f'Insufficient funds for withdrawal of {amt}')

    def get_balance(self):
        return self._bal

    def __str__(self):
        return f'Account with balance {self._bal}'
```

Note: The Python 3.7+ source code for the above version of `Account` is available in linked file `account5.py`.

We can see the effects of the decorator in the following REPL session.

```
>>> acct = Account()
>>> str(acct)
Get: _bal
'Account with balance 0'
>>> acct.deposit(100)
Get: deposit
Get: _bal
>>> str(acct)
Get: _bal
'Account with balance 100'
>>> acct.withdraw(60)
Get: withdraw
Get: _bal
Get: _bal
>>> str(acct)
Get: _bal
'Account with balance 40'
>>> acct.get_balance()
Get: get_balance
Get: _bal
40
>>> str(acct)
'Account with balance 40'
```

Note that both calls to the methods and the accesses to the “private” data attribute `_bal` are shown. (If we want to exclude accesses to the private instance variables, we can modify `debugattr` to exclude attributes whose names begin with a single underscore.)

9.4 Class Hierarchy Debugging

9.4.1 Motivating example

Now let's set up class-level debugging on the inheritance hierarchy `P` example from Chapter 7.

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

@debugmethods
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()}'

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

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

Note: The Python 3.7+ source code for the above version of the `P` class hierarchy is available in linked file `inherit2.py`.

So, we have another occurrence of code redundancy that we saw at the class level in the previous section. Let's see if we can DRY out the code more.

To do this, the program needs to process the whole class hierarchy rooted at class `P`. Let's review the nature of the Python object model to see how to do this.

9.4.2 Review of objects and types

In Chapters 5-7 of these notes, we learned:

- All Python values are objects.
- All objects have types.
- A class defines a new type.

- A class is a callable (i.e., function) that creates instances; the class is the type of the instances it creates. Hence, in some sense, a class *is a type* consisting of its potential instances and the operations it defines.
- A class itself is an object. It is an instance of other classes. Thus it *has a type*.
- The built-in class `type` is the root class (i.e., top-level metaclass) for all other classes (i.e., types). When a program invokes `type` as a constructor, it creates a new type (i.e., class) object.
- Classes may inherit (i.e., be a subclass of) other classes.
- The built-in class `object` is the root class for all other top-level user-defined and built-in classes.

TODO: Maybe repeat diagram below here.

Note: See the diagram in Figure 7-1 from Chapter 7.

The following Python REPL session illustrates these concepts.

```
>>> class PP:
...     pass
...
>>> class CC(PP):
...     pass
...
>>> PP
<class '__main__.PP'>
>>> type(PP)
<class 'type'>
>>> issubclass(P,object)
True
>>> CC
<class '__main__.CC'>
>>> type(CC)
<class 'type'>
>>> issubclass(CC,PP)
True
>>> x = PP()
>>> x
<__main__.PP object at 0x10cd3d048>
>>> isinstance(x,PP)
True
>>> type(x)
<class '__main__.PP'>
>>> type
<class 'type'>
>>> type(type)
```

```
<class 'type'>
>>> issubclass(type,object)
True
>>> object
<class 'object'>
>>> type(object)
<class 'type'>
```

9.4.3 Class definition process

Now let's examine how the Python interpreter elaborates class definitions at runtime. Consider the class `MyClass` defined as follows:

```
class MyClass(Parent):
    def __init__(self, id):
        self.id = id
    def hello(self):
        print(f'Hello from MyClass.hello, id = {self.id}')"
```

This class definition has three components.

- Name: *"MyClass"*
- Base classes: `(Parent,)`
- Functions: `(___init___, hello)`

The interpreter takes the following steps during class definition.

1. It isolates the body of the class. (Note the multiline string below.)

   ```
   body = '''
   def __init__(self, myid):
       self.myid = myid
   def hello(self):
       print(f'Hello from MyClass.hello, myid = {self.myid}')
   '''
   ```

2. It creates the class dictionary.

   ```
   clsdict = type.__prepare__('MyClass', (Parent,))
   ```

   Method `type.__prepare__` is a class method on the root metaclass `type`. In the process of creating the new class object for a class, the interpreter calls the `__prepare__` method before it calls the `__new__` method on `type` [31:701–3].

   In addition to metaclass argument (i.e., `type`), the `__prepare__` class method takes two additional arguments:

   - the *name* of the *class* being created (e.g., *'MyClass'* above)
   - a *tuple* of the one or more base classes (e.g., `(Parent,)` above)

   Method `__prepare__` returns a dictionary that can be subsequently passed to the `__new__` and `__init__` methods. This dictionary serves as the local namespace for the statements in the class body.

3. It executes the `body` dynamically (using `exec`) in the current global namespace (returned by the call `globals()`) and the local namespace defined by the class dictionary `clsdict`.

   ```
   exec(body, globals(), clsdict)
   ```

   This step populates `clsdict`.

   ```
   >>> clsdict
   {'__init__': <function __init__ at 0x10cc21ea0>,
    'hello': <function hello at 0x10d2b9bf8>}
   ```

4. It constructs the class object using its name, its base classes, and the dictionary populated in the previous step.

   ```
   >>> MyClass = type('MyClass', (Parent,), clsdict)
   >>> MyClass
   <class '__main__.MyClass'>
   >>> mc = MyClass('Conrad')
   <__main__.MyClass object at 0x100f96c50>
   >>> mc.myid
   Conrad
   >>> mc.hello()
   Hello from MyClass.hello, myid = Conrad
   ```

   The call `type('MyClass', (Parent,), clsdict)` constructs an instance of metaclass `type` with name `MyClass`, superclass `Parent`, and object dictionary `clsdict`. This is the class object for `MyClass`.

Note: The Python 3.7+ source code for the above creation of class `MyClass` is available in linked file `MyClass1.py`.

9.4.4 Changing the metaclass

A Python class definition has a keyword parameter named `metaclass` whose default value is `type`. So the parent class `P` from the motivating example for this section is equivalent to the following.

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

This keyword parameter sets the class for creating the new type for the class. Although the default is `type`, we can change it to some other metaclass.

To define a new metaclass, we typically define a type that inherits from `type` and gives a new definition for one or both of the special methods `__new__` and `__init__`.

```
class mytype(type):
    def __new__(cls, name, bases, clsdict):
        # possible preprocessing of arguments
        clsobj = super().__new__(cls, name, bases, clsdict)
        # possible postprocessing of object
        return clsobj
```

The special method `__new__` allocates memory, constructs a new instance (i.e., object), and then returns it. The interpreter passes this new instance to special method `__init__`, which initializes the new instance variables.

We do not normally override `__new__`, but in a metaclass we may want to do some additional work either before or after the basic construction processing.

A metaclass can access information about a class definition at the time the class is defined. It can inspect the data and, if needed, modify the data.

Given the above definition, we can use the new metaclass as follows:

```
class P(metaclass=mytype):
    ...
```

9.4.5 Debugging using a metaclass

Now we have the tools we need to remove the code redundancy from the motivating example. We can introduce the *metaclass* shown in the example below.

```
class debugmeta(type):
    def __new__(cls, clsname, bases, clsdict):
        clsobj = super().__new__(cls,clsname,bases,clsdict)  #1
        clsobj = debugmethods(clsobj)                        #2
        return clsobj                                        #3
```

The approach above:

1. creates the class normally (using `super().__new__`)
2. immediately wraps the class object with the class-level debug decorator `debugmethods` we developed previously
3. then returns the wrapped class object

Given the above metaclass definition, we can apply it to the inheritance example as sketched below.

```
class P(metaclass = debugmeta):
    ...
```

```
class C(P):
    ...
class D(P):
    ...
class G(C):
    ...
```

Note: The Python 3.7+ source code for the above version of the `P` class hierarchy is available in linked file `inherit3.py`.

Now consider a Python REPL session using the above code with the custom metaclass.

```
>>> from inherit3 import *
>>> type(P)
<class 'inherit3.debugmeta'>
>>> issubclass(P,object)
True
>>> type(C)
<class 'inherit3.debugmeta'>
>>> issubclass(C,P)
True
>> type(G)
<class 'inherit3.debugmeta'>
>>> issubclass(G,C)
True
>>> issubclass(G,P)
True
>>> p1 = P()
P.__init__
>>> type(p1)
<class 'inherit3.P'>
>>> c1 = C()
P.__init__
>>> type(c1)
<class 'inherit3.C'>
>>> g1 = G()
P.__init__
>>> type(g1)
<class 'inherit3.C'>
>>> p1.process()
P.process
'Process at parent P level'
>>> c1.process()
C.process
P.process
'Process at child C level \n  Process at parent P level'
```

```
>>> g1.process()
G.process
'Process at grandchild G level'
```

9.4.6 Why metaclasses?

As we have seen, we can transform a class in similar ways using either a class decorator or a metaclass.

Given that a class decorator is easier to set up and apply, when and why should we use a metaclass?

One advantage to metaclasses is that they can propagate down class <hierarchies. Consider our motivating example again.

```
class P(metaclass = debugmeta):
    ...
class C(P):   # metaclass = debugmeta
    ...
class D(P):   # metaclass = debugmeta
    ...
class G(C):   # metaclass = debugmeta
    ...
```

As we can see in the REPL session output in the previous subsection, use of the metaclass in parent class `P` is passed down automatically to all its descendants. No changes are needed to the descendant classes.

In some sense, the metaclass mutates the DNA of the parent class and that mutation is passed on to the children. In this example, debugging is applied across the entire hierarchy. The code is kept DRY.

9.5 What Next?

In this case study, we used Python metaprogramming facilities to debug successively larger program units. But regardless of the level, the method mostly involved wrapping and rewriting the program units.

- We used function decorators to wrap and rewrite functions.
- We used class decorators to wrap and rewrite classes.
- We used metaclasses to wrap and rewrite class hierarchies.

So far, we have mostly used "classic" metaprogramming techniques that were available in Python 2 with only a few Python 3 features.

In the coming chapters, we use more advanced features of Python 3. (These chapters are planned but not yet drafted.)

9.6 Chapter Source Code

TODO

9.7 Exercises

TODO

9.8 Acknowledgements

I originally developed these notes in Spring 2018 for use in CSci 658 Software Language Engineering [9]. The Spring 2018 version used Python 3.6. I updated them some for use in CSci 556 Multiparadigm Programming, which used Python 3.7.

The overall set of notes on Python 3 Reflexive Metaprogramming is inspired by David Beazley's Python 3 Metaprogramming tutorial from PyCon'2013 [3]. In particular, some chapters adapt Beazley's examples. Beazley's tutorial draws on material from his and Brian K. Jones' book *Python Cookbook* [4].

In particular, this chapter adapts Beazley's `debugly` example presentation from his Python 3 Metaprogramming tutorial at PyCon'2013 [3].

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 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.

9.9 Terms and Concepts

TODO

9.10 References

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

[2] Harold Abelson and Gerald Jockay 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] Richard Bird. 1998. *Introduction to functional programming using Haskell* (Second ed.). Prentice Hall, Englewood Cliffs, New Jersey, USA.

[6] Kathryn Heninger Britton, R. Alan Parker, and David L. Parnas. 1981. A procedure for designing abstract interfaces for device interface modules. In *Proceedings of the 5th international conference on software engineering*, IEEE, San Diego, California, USA, 195–204.

[7] Timothy Budd. 2000. *Understanding object-oriented programming with Java* (Updated ed.). Addison-Wesley, Boston, Massachusetts, USA.

[8] Timothy Budd. 2002. *An introduction to object oriented programming* (Third ed.). Addison-Wesley, Boston, Massachusetts, USA. Retrieved from https://web.engr.oregonstate.edu/~budd/Books/oopintro3e/info/toc.pdf

[9] 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

[10] 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

[11] H. Conrad Cunningham. 2019. *Type system concepts.* University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/TypeConcepts/TypeSystemConcepts.html

[12] H. Conrad Cunningham. 2022. *Notes on domain-specific languages.* University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/DSLs/NotesDSLs.html

[13] 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

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

[15] Martin Fowler and Rebecca Parsons. 2010. *Domain specific languages.* Addison-Wesley, Boston, Massachusetts, USA.

[16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design patterns: Elements of reusable object-oriented software.* Addison-Wesley, Boston, Massachusetts, USA.

[17] Cay S. Horstmann. 1995. *Mastering object-oriented design in C++.* Wiley, Indianapolis, Indiana, USA.

[18] Cay S. Horstmann and Gary Cornell. 1999. *Core Java 1.2: Volume I—Fundamentals.* Prentice Hall, Englewood Cliffs, New Jersey, USA.

[19] Paul Hudak. 1989. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys* 21, 3 (1989), 359–411.

[20] Andrew Hunt and David Thomas. 1999. *The pragmatic programme.* Addison-Wesley, Boston Massachusetts, USA.

[21] Roberto Ierusalimschy. 2013. *Programming in Lua* (Third ed.). Lua.org, Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Brazil.

[22] Brian W. Kernighan. 1984. *PIC—a graphics language for typesetting, revised user manual.* Bell Laboratories, Computing Science, Murray Hill, New Jersey, USA. Retrieved from http://doc.cat-v.org/unix/v8/picmemo.pdf

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

[24] Barbara Liskov. 1987. Keynote address—Data abstraction and hierarchy. In *Proceedings on object-oriented programming systems, languages, and applications (OOPSLA '87): addendum*, ACM, Orlando, Florida, USA, 17–34.

[25] Bertrand Meyer. 1997. *Object-oriented program construction* (Second ed.). Prentice Hall, Englewood Cliffs, New Jersey, USA.

[26] Martin Odersky, Lex Spoon, and Bill Venners. 2008. *Programming in Scala* (First ed.). Artima, Inc., Walnut Creek, California, USA.

[27] David L. Parnas. 1972. On the criteria to be used in decomposing systems into modules. *Communications of the ACM* 15, 12 (December 1972), 1053–1058.

[28] David L. Parnas. 1976. On the design and development of program families. *IEEE Transactions on Software Engineering* SE-2, 1 (1976), 1–9.

[29] Benjamin C. Pierce. 2002. *Types and programming language.* MIT Press, Cambridge, Massachusetts, USA.

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

[31] Luciano Ramalho. 2013. *Fluent Python: Clear, concise, and effective programming.* O'Reilly Media, Sebastopol, California, USA.

[32] Eric S. Raymond. 1995. Making pictures with GNU PIC. Retrieved from https://lists.gnu.org/r/groff/2011-08/pdfbLauVhlfQs.pdf

[33] Michael L. Scott. 2015. *Programming language pragmatics* (Third ed.). Morgan Kaufmann, Waltham, Massachusetts, USA.

[34] Robert W. Sebesta. 1993. *Concepts of programming languages* (Second ed.). Benjamin/Cummings, Boston, Massachusetts, USA.

[35] StackOverflow. 2012. Meaning of @classmethod and @staticmethod for beginner? Retrieved from https://stackoverflow.com/questions/12179271/meaning-of-classmethod-and-staticmethod-for-beginner

[36] Richard M. Stallman and Zachary Weinberg. 2022. The c preprocessor. Retrieved from https://gcc.gnu.org/onlinedocs/cpp/

[37] Simon Thompson. 1996. *Haskell: The craft of programming* (First ed.). Addison-Wesley, Boston, Massachusetts, USA.

[38] Bill Venners. 2003. Orthogonality and the DRY principle: Interview of Dave Thomas. Retrieved from https://www.artima.com/intv/dry.html

[39] Wikpedia: The Free Encyclopedia. 2022. Liskov substitution principle. Retrieved from https://en.wikipedia.org/wiki/Liskov_substitution_principle

[40] Wikpedia: The Free Encyclopedia. 2022. Polymorphism (computer science). Retrieved from https://en.wikipedia.org/wiki/Polymorphism_(computer_science)

[41] Wikpedia: The Free Encyclopedia. 2022. Function overloading. Retrieved from https://en.wikipedia.org/wiki/Function_overloading

[42] Wikpedia: The Free Encyclopedia. 2022. Ad hoc polymorphism. Retrieved from https://en.wikipedia.org/wiki/Ad_hoc_polymorphism

[43] Wikpedia: The Free Encyclopedia. 2022. Parametric polymophism. Retrieved from https://en.wikipedia.org/wiki/Parametric_polymorphism

[44] Wikpedia: The Free Encyclopedia. 2022. Subtyping. Retrieved from https://en.wikipedia.org/wiki/Subtyping

[45] Wikpedia: The Free Encyclopedia. 2022. Metaprogramming. Retrieved from https://en.wikipedia.org/wiki/Metaprogramming

[46] Wikpedia: The Free Encyclopedia. 2022. Reflective programming. Retrieved from https://en.wikipedia.org/wiki/Reflective_programming

[47] Wikpedia: The Free Encyclopedia. 2022. Don't repeat yourself. Retrieved from https://en.wikipedia.org/wiki/Don't_repeat_yourself