California State University Fullerton

CPSC-223P

Python Programming

Stephen T. May

Python Tutorial

Section 9

Classes

https://docs.python.org/release/3.9.6/tutorial/index.html

Slide Notes

- Command typed at the Linux command prompt ($)

```
$ python3.9
```

- Command typed at the Python interpreter command prompt (>>>)

```
>>> Ctrl-D
```

- Python source code

```
print(“Hello world!”)
```

- Mixed example

```
>>> the_world_is_flat = True
>>> if the_world_is_flat:
...     print("Be careful not to fall off!")
...
Be careful not to fall off!
```

9. Classes

- Classes provide a means of bundling data and functionality together
- Creating a new class creates a new type of object, allowing new instances of that type to be made
- Each class instance can have attributes attached to it for maintaining its state
- Class instances can also have methods (defined by its class) for modifying its state
- Python classes provide all the standard features of Object Oriented Programming:
  - the class inheritance mechanism allows multiple base classes
  - a derived class can override any methods of its base class or classes
  - and a method can call the method of a base class with the same name

9.1. A Word About Names and Objects

- Objects have individuality, and multiple names can be bound to the same object
- This is known as aliasing in other languages
- This is usually not appreciated on a first glance at Python, and can be safely ignored when dealing with immutable basic types (numbers, strings, tuples)
- However, aliasing has a possibly surprising effect on the semantics of Python code involving mutable objects such as lists, dictionaries, and most other types
- This is usually used to the benefit of the program, since aliases behave like pointers in some respects
- For example, passing an object is cheap since only a pointer is passed by the implementation
- If a function modifies an object passed as an argument, the caller will see the change
- This eliminates the need for two different argument passing mechanisms as in C++

9.2. Python Scopes and Namespaces

- A *namespace* is a mapping from names to objects
- Most namespaces are currently implemented as Python dictionaries
- Examples of namespaces are:
  - The set of built-in names (containing functions such as `abs()`, and built-in exception names)
  - The global names in a module
  - The local names in a function invocation
- In a sense the set of attributes of an object also form a namespace
- There is absolutely no relation between names in different namespaces
- For instance, two different modules may both define a function `maximize` without confusion — users of the modules must prefix it with the module name
- The word *attribute* is used for any name following a dot — for example, in the expression `z.real`, `real` is an attribute of the object `z`
- Attributes may be *read-only* or *writable*
- In the latter case, assignment to attributes is possible
- Module attributes are writable: you can write `modname.the_answer = 42`
- Writable attributes may also be deleted with the `del` statement
- For example, `del modname.the_answer` will remove the attribute the_`answer` from the object named by `modname`

9.2. Python Scopes and Namespaces (cont.)

- Namespaces are created at different moments and have different lifetimes
- The namespace containing the *built-in names* is created when the Python interpreter starts up, and is never deleted
- The *global namespace* for a module is created when the module definition is read in; normally, module namespaces also last until the interpreter quits
- The statements executed by the *top-level invocation of the interpreter*, either read from a script file or interactively, are considered part of a module called __main__, so they have their own global namespace
- The *local namespace for a function* is created when the function is called, and deleted when the function returns or raises an exception that is not handled within the function
- *Recursive invocations* each have their own local namespace

9.2. Python Scopes and Namespaces (cont.)

- A *scope* is a textual region of a Python program where a namespace is directly accessible
- *Directly accessible* here means that an unqualified reference to a name attempts to find the name in the namespace
- Although scopes are determined statically, they are used dynamically
- At any time during execution, there are 3 or 4 nested scopes whose namespaces are directly accessible:
  - The *innermost scope*, which is searched first, contains the local names
  - The *scopes of any enclosing functions*, which are searched starting with the nearest enclosing scope, contains non-local, but also non-global names
  - The *next-to-last scope* contains the current module's global names
  - The *outermost scope* (searched last) is the namespace containing built-in names
- If a name is declared `global`, then all references and assignments go directly to the middle scope containing the module's global names
- To rebind variables found outside of the innermost scope, the `nonlocal` statement can be used
- If not declared `nonlocal`, those variables are read-only
- An attempt to write to such a variable will simply create a new local variable in the innermost scope, leaving the identically named outer variable unchanged
- Usually, the local scope references the local names of the current function
- Outside of functions, the local scope references the same namespace as the global scope: the module's namespace
- Class definitions place yet another namespace in the local scope

9.2. Python Scopes and Namespaces (cont.)

- If no `global` or `nonlocal` statement is in effect, assignments to names always go into the innermost scope
- Assignments do not copy data — they just bind names to objects
- The same is true for deletions: the statement `del x` removes the binding of `x` from the namespace referenced by the local scope
- In fact, all operations that introduce new names use the local scope
- `import` statements and function definitions bind the module or function name in the local scope
- The `global` statement can be used to indicate that particular variables live in the *global scope* and should be re-bound there
- The `nonlocal` statement indicates that particular variables live in an *enclosing scope* and should be re-bound there

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

9.3.1. Class Definition Syntax

- *Class definitions*, like function definitions must be executed before they have any effect
- In practice, the statements inside a `class` definition will usually be function definitions, but other statements are allowed, and sometimes useful
- The function definitions inside a class normally have a specific form of argument list, dictated by the calling conventions for methods
- When a class definition is entered, a new namespace is created, and used as the local scope — all assignments to local variables go into this new namespace
- Function definitions bind the name of the new function here
- When a class definition is left normally (via the end), a *class object* is created
- This is basically a wrapper around the contents of the namespace created by the class definition
- The original local scope (the one in effect just before the class definition was entered) is reinstated, and the class object is bound here to the class name given in the class definition header

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

9.3.2. Class Objects

- Class objects support two kinds of operations: *attribute references* and *instantiation*
- *Attribute references* use the standard syntax used for all attribute references in Python: `obj.name`
- Valid attribute names are all the names that were in the class's namespace when the class object was created
- `MyClass.i` and `MyClass.f` are valid attribute references, returning an integer and a function object, respectively
- Class attributes can also be assigned to, so you can change the value of `MyClass.i` by assignment

```
class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
        return 'hello world'
```

9.3.2. Class Objects (cont.)

- *Class instantiation* uses function notation
- The example creates a new instance of the class and assigns this object to the local variable `x`

```
class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
        return 'hello world'
```

```
x = MyClass()
```

- Many classes like to create objects with instances customized to a specific initial state
- When a class defines an `__init__()` method, class instantiation automatically invokes `__init__()` for the newly-created class instance
- The `__init__()` method may have arguments for greater flexibility
- Arguments given to the class instantiation operator are passed on to `__init__()`

```
def __init__(self):
    self.data = []
```

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

9.3.3. Instance Objects

- The only operations understood by instance objects are *attribute references*
- There are two kinds of valid attribute names: *data attributes* and *methods*
- *Data attributes* correspond o “data members” in C++
- Data attributes need not be declared; like local variables, they spring into existence when they are first assigned to
- A *method* is a function that “belongs to” an object
- Valid method names of an instance object depend on its class
- By definition, all attributes of a class that are function objects define corresponding methods of its instances
- So in our example, `x.f` is a valid method reference, since `MyClass.f` is a function, but `x.i` is not, since `MyClass.i` is not
- But `x.f` is not the same thing as `MyClass.f` — it is a method object, not a function object

```
class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
        return 'hello world'
```

```
x = MyClass()
```

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print(x.counter)
del x.counter
```

9.3.4. Method Objects

- Usually, a method is called right after it is bound
- In the `MyClass` example, this will return the string 'hello world'

```
class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
        return 'hello world'
```

```
x.f()
```

- However, it is not necessary to call a method right away
- `x.f` is a method object, and can be stored away and called at a later time
- The example will continue to print hello world until the end of time

```
xf = x.f
while True:
    print(xf())
```

- The special thing about methods is that the instance object is passed as the first argument of the function
- The call `x.f()` is exactly equivalent to `MyClass.f(x)`

9.3.5. Class and Instance Variables

- *Class variables* are for attributes and methods shared by all instances of the class
- *Instance variables* are for data unique to each instance

```
class Dog:

    kind = 'canine'         # class variable shared by all
instances

    def __init__(self, name):
        self.name = name    # instance variable unique to each
instance

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind                  # shared by all dogs
'canine'
>>> e.kind                  # shared by all dogs
'canine'
>>> d.name                  # unique to d
'Fido'
>>> e.name                  # unique to e
'Buddy'
```

9.3.5. Class and Instance Variables (cont.)

- Shared data can have possibly surprising effects with involving mutable objects such as lists and dictionaries
- The tricks list in the following code should not be used as a class variable because just a single list would be shared by all Dog instances

```
class Dog:

    tricks = []             # mistaken use of a class variable

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks                # unexpectedly shared by all dogs
['roll over', 'play dead']
```

- The correct design of the class should use an instance variable instead

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']
```

9.4. Random Remarks

- If the same attribute name occurs in both an instance and in a class, then attribute lookup prioritizes the instance

```
>>> class Warehouse:
        purpose = 'storage'
        region = 'west'

>>> w1 = Warehouse()
>>> print(w1.purpose, w1.region)
storage west
>>> w2 = Warehouse()
>>> w2.region = 'east'
>>> print(w2.purpose, w2.region)
storage east
```

- Methods may call other methods by using method attributes of the self argument

```
class Bag:
    def __init__(self):
        self.data = []

    def add(self, x):
        self.data.append(x)

    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

9.5. Inheritance

- The name BaseClassName must be defined in a scope containing the derived class definition
- In place of a base class name, other arbitrary expressions are also allowed
- This can be useful, for example, when the base class is defined in another module
- Method references are resolved as follows
  - the corresponding class attribute is searched
  - descending down the chain of base classes if necessary
  - and the method reference is valid if this yields a function object
- Derived classes may override methods of their base classes
- Python has two built-in functions that work with inheritance
  - Use `isinstance()` to check an instance's type: `isinstance(obj, int)` will be True only if `obj.__class__` is int or some class derived from int
  - Use `issubclass()` to check class inheritance: `issubclass(bool, int)` is True since bool is a subclass of int
  - However, `issubclass(float, int)` is False since float is not a subclass of int

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

```
class DerivedClassName(modname.BaseClassName):
```

9.5.1. Multiple Inheritance

- Python supports a form of multiple inheritance as well
- For most purposes, in the simplest cases, you can think of the search for attributes inherited from
  - A parent class as depth-first
  - Left-to-right
  - Not searching twice in the same class where there is an overlap in the hierarchy
- Thus, if an attribute is not found in `DerivedClassName`
  - It is searched for in `Base1`
  - Then (recursively) in the base classes of `Base1`
  - And if it was not found there
  - It was searched for in `Base2`
  - And so on

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

9.6. Private Variables

- "Private" instance variables that cannot be accessed except from inside an object don't exist in Python
- However, there is a convention that is followed by most Python code: a name prefixed with an underscore (e.g. _spam) should be treated as a non-public part of the API
- Since there is a valid use-case for class-private members (namely to avoid name clashes of names with names defined by subclasses), there is limited support for such a mechanism, called name mangling
- Any identifier of the form __spam (at least two leading underscores, at most one trailing underscore) is textually replaced with _classname__spam, where classname is the current class name with leading underscore(s) stripped
- Name mangling is helpful for letting subclasses override methods without breaking intraclass method calls

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

    __update = update   # private copy of original update() method

class MappingSubclass(Mapping):

    def update(self, keys, values):
        # provides new signature for update()
        # but does not break __init__()
        for item in zip(keys, values):
            self.items_list.append(item)
```

9.7. Odds and Ends

- Sometimes it is useful to have a data type similar to the C "struct", bundling together a few named data items
- An empty class definition will do nicely
- A piece of Python code that expects a particular abstract data type can often be passed a class that emulates the methods of that data type instead
- For instance, if you have a function that formats some data from a file object, you can define a class with methods `read()` and `readline()` that get the data from a string buffer instead, and pass it as an argument

```
class Employee:
    pass

john = Employee()  # Create an empty employee record

# Fill the fields of the record
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

9.8. Iterators

- Most container objects can be looped over using a `for` statement
- This style of access is clear, concise, and convenient
- Behind the scenes, the `for` statement calls `iter()` on the container object
- The function returns an iterator object that defines the method `__next__()` which accesses elements in the container one at a time
- When there are no more elements, `__next__()` raises a `StopIteration` exception which tells the for loop to terminate
- You can call the `__next__()` method using the `next()` built-in function

```
for element in [1, 2, 3]:
    print(element)
for element in (1, 2, 3):
    print(element)
for key in {'one':1, 'two':2}:
    print(key)
for char in "123":
    print(char)
for line in open("myfile.txt"):
    print(line, end='')
```

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> next(it)
'a'
>>> next(it)
'b'
>>> next(it)
'c'
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    next(it)
StopIteration
```

9.8. Iterators (cont.)

- Having seen the mechanics behind the iterator protocol, it is easy to add iterator behavior to your classes
- Define an `__iter__()` method which returns an object with a `__next__()` method
- If the class defines `__next__()`, then `__iter__()` can just return `self`

```
class Reverse:
    """Iterator for looping over a sequence backwards."""
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __iter__(self):
        return self

    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
```

```
>>> rev = Reverse('spam')
>>> iter(rev)
<__main__.Reverse object at 0x00A1DB50>
>>> for char in rev:
...     print(char)
...
m
a
p
s
```

9.9. Generators

- *Generators* are a simple and powerful tool for creating iterators
- They are written like regular functions but use the `yield` statement whenever they want to return data
- Each time `next()` is called on it, the generator resumes where it left off
- What makes generators so compact is that the `__iter__()` and `__next__()` methods are created automatically
- Another key feature is that the local variables and execution state are automatically saved between calls
- In addition to automatic method creation and saving program state, when generators terminate, they automatically raise `StopIteration`
- In combination, these features make it easy to create iterators with no more effort than writing a regular function

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]
```

```
>>> for char in reverse('golf'):
...     print(char)
...
f
l
o
g
```

9.10. Generator Expressions

- Some simple generators can be coded succinctly as expressions using a syntax similar to list comprehensions but with parentheses instead of square brackets
- These expressions are designed for situations where the generator is used right away by an enclosing function
- *Generator expressions* are more compact but less versatile than full generator definitions and tend to be more memory friendly than equivalent list comprehensions

```
>>> sum(i*i for i in range(10))                 # sum of squares
285

>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvec, yvec))         # dot product
260

>>> unique_words = set(word for line in page  for word in
line.split())

>>> valedictorian = max((student.gpa, student.name) for student in
graduates)

>>> data = 'golf'
>>> list(data[i] for i in range(len(data)-1, -1, -1))
['f', 'l', 'o', 'g']
```