# 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.

Compared with other programming languages, Python’s class mechanism adds classes with a minimum of new syntax and semantics. It is a mixture of the class mechanisms found in C++ and Modula-3. 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. Objects can contain arbitrary amounts and kinds of data. As is true for modules, classes partake of the dynamic nature of Python: they are created at runtime, and can be modified further after creation.

In C++ terminology, normally class members (including the data members) are public (except see below [Private Variables](https://docs.python.org/3/tutorial/classes.html#tut-private)), and all member functions are virtual. As in Modula-3, there are no shorthands for referencing the object’s members from its methods: the method function is declared with an explicit first argument representing the object, which is provided implicitly by the call. As in Smalltalk, classes themselves are objects. This provides semantics for importing and renaming. Unlike C++ and Modula-3, built-in types can be used as base classes for extension by the user. Also, like in C++, most built-in operators with special syntax (arithmetic operators, subscripting etc.) can be redefined for class instances.

(Lacking universally accepted terminology to talk about classes, I will make occasional use of Smalltalk and C++ terms. I would use Modula-3 terms, since its object-oriented semantics are closer to those of Python than C++, but I expect that few readers have heard of it.)

## 9.1. A Word About Names and Objects

Objects have individuality, and multiple names (in multiple scopes) 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; and 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 Pascal.

## 9.2. Python Scopes and Namespaces

Before introducing classes, I first have to tell you something about Python’s scope rules. Class definitions play some neat tricks with namespaces, and you need to know how scopes and namespaces work to fully understand what’s going on. Incidentally, knowledge about this subject is useful for any advanced Python programmer.

Let’s begin with some definitions.

A namespace is a mapping from names to objects. Most namespaces are currently implemented as Python dictionaries, but that’s normally not noticeable in any way (except for performance), and it may change in the future. Examples of namespaces are: the set of built-in names (containing functions such as [abs()](https://docs.python.org/3/library/functions.html#abs), and built-in exception names); the global names in a module; and the local names in a function invocation. In a sense the set of attributes of an object also form a namespace. The important thing to know about namespaces is that 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.

By the way, I use the word attribute for any name following a dot — for example, in the expression z.real, real is an attribute of the object z. Strictly speaking, references to names in modules are attribute references: in the expression modname.funcname, modname is a module object and funcname is an attribute of it. In this case there happens to be a straightforward mapping between the module’s attributes and the global names defined in the module: they share the same namespace! [1](https://docs.python.org/3/tutorial/classes.html#id2)

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](https://docs.python.org/3/reference/simple_stmts.html#del) statement. For example, del modname.the\_answer will remove the attribute the\_answer from the object named by modname.

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\_\_](https://docs.python.org/3/library/__main__.html#module-__main__), so they have their own global namespace. (The built-in names actually also live in a module; this is called [builtins](https://docs.python.org/3/library/builtins.html" \l "module-builtins" \o "builtins: The module that provides the built-in 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. (Actually, forgetting would be a better way to describe what actually happens.) Of course, recursive invocations each have their own local namespace.

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](https://docs.python.org/3/reference/simple_stmts.html#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 (textually) current function. Outside 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.

It is important to realize that scopes are determined textually: the global scope of a function defined in a module is that module’s namespace, no matter from where or by what alias the function is called. On the other hand, the actual search for names is done dynamically, at run time — however, the language definition is evolving towards static name resolution, at “compile” time, so don’t rely on dynamic name resolution! (In fact, local variables are already determined statically.)

A special quirk of Python is that – if no [global](https://docs.python.org/3/reference/simple_stmts.html#global) or [nonlocal](https://docs.python.org/3/reference/simple_stmts.html#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: in particular, [import](https://docs.python.org/3/reference/simple_stmts.html#import) statements and function definitions bind the module or function name in the local scope.

The [global](https://docs.python.org/3/reference/simple_stmts.html#global) statement can be used to indicate that particular variables live in the global scope and should be rebound there; the [nonlocal](https://docs.python.org/3/reference/simple_stmts.html#nonlocal) statement indicates that particular variables live in an enclosing scope and should be rebound there.

### **9.2.1. Scopes and Namespaces Example**

This is an example demonstrating how to reference the different scopes and namespaces, and how [global](https://docs.python.org/3/reference/simple_stmts.html#global) and [nonlocal](https://docs.python.org/3/reference/simple_stmts.html#nonlocal) affect variable binding:

**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)

The output of the example code is:

After local assignment: test spam

After nonlocal assignment: nonlocal spam

After global assignment: nonlocal spam

In global scope: global spam

Note how the local assignment (which is default) didn’t change scope\_test's binding of spam. The [nonlocal](https://docs.python.org/3/reference/simple_stmts.html#nonlocal) assignment changed scope\_test's binding of spam, and the [global](https://docs.python.org/3/reference/simple_stmts.html#global) assignment changed the module-level binding.

You can also see that there was no previous binding for spam before the [global](https://docs.python.org/3/reference/simple_stmts.html#global) assignment.

## 9.3. A First Look at Classes

Classes introduce a little bit of new syntax, three new object types, and some new semantics.

### **9.3.1. Class Definition Syntax**

The simplest form of class definition looks like this:

**class** **ClassName**:

<statement-1>

.

.

.

<statement-N>

Class definitions, like function definitions ([def](https://docs.python.org/3/reference/compound_stmts.html#def) statements) must be executed before they have any effect. (You could conceivably place a class definition in a branch of an [if](https://docs.python.org/3/reference/compound_stmts.html#if) statement, or inside a function.)

In practice, the statements inside a class definition will usually be function definitions, but other statements are allowed, and sometimes useful — we’ll come back to this later. The function definitions inside a class normally have a peculiar form of argument list, dictated by the calling conventions for methods — again, this is explained later.

When a class definition is entered, a new namespace is created, and used as the local scope — thus, all assignments to local variables go into this new namespace. In particular, 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; we’ll learn more about class objects in the next section. 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 (ClassName in the example).

### **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. So, if the class definition looked like this:

**class** **MyClass**:

*"""A simple example class"""*

i = 12345

**def** f(self):

**return** 'hello world'

then 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. \_\_doc\_\_ is also a valid attribute, returning the docstring belonging to the class: "A simple example class".

Class instantiation uses function notation. Just pretend that the class object is a parameterless function that returns a new instance of the class. For example (assuming the above class):

x = MyClass()

creates a new instance of the class and assigns this object to the local variable x.

The instantiation operation (“calling” a class object) creates an empty object. Many classes like to create objects with instances customized to a specific initial state. Therefore a class may define a special method named \_\_init\_\_(), like this:

**def** \_\_init\_\_(self):

self.data = []

When a class defines an \_\_init\_\_() method, class instantiation automatically invokes \_\_init\_\_() for the newly-created class instance. So in this example, a new, initialized instance can be obtained by:

x = MyClass()

Of course, the \_\_init\_\_() method may have arguments for greater flexibility. In that case, arguments given to the class instantiation operator are passed on to \_\_init\_\_(). For example,

>>>

**>>> 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**

Now what can we do with 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 to “instance variables” in Smalltalk, and to “data members” in C++. Data attributes need not be declared; like local variables, they spring into existence when they are first assigned to. For example, if x is the instance of MyClass created above, the following piece of code will print the value 16, without leaving a trace:

x.counter = 1

**while** x.counter < 10:

x.counter = x.counter \* 2

print(x.counter)

**del** x.counter

The other kind of instance attribute reference is a method. A method is a function that “belongs to” an object. (In Python, the term method is not unique to class instances: other object types can have methods as well. For example, list objects have methods called append, insert, remove, sort, and so on. However, in the following discussion, we’ll use the term method exclusively to mean methods of class instance objects, unless explicitly stated otherwise.)

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.

### **9.3.4. Method Objects**

Usually, a method is called right after it is bound:

x.f()

In the MyClass example, this will return the string 'hello world'. 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. For example:

xf = x.f

**while** **True**:

print(xf())

will continue to print hello world until the end of time.

What exactly happens when a method is called? You may have noticed that x.f() was called without an argument above, even though the function definition for f() specified an argument. What happened to the argument? Surely Python raises an exception when a function that requires an argument is called without any — even if the argument isn’t actually used…

Actually, you may have guessed the answer: the special thing about methods is that the instance object is passed as the first argument of the function. In our example, the call x.f() is exactly equivalent to MyClass.f(x). In general, calling a method with a list of n arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method’s instance object before the first argument.

If you still don’t understand how methods work, a look at the implementation can perhaps clarify matters. When a non-data attribute of an instance is referenced, the instance’s class is searched. If the name denotes a valid class attribute that is a function object, a method object is created by packing (pointers to) the instance object and the function object just found together in an abstract object: this is the method object. When the method object is called with an argument list, a new argument list is constructed from the instance object and the argument list, and the function object is called with this new argument list.

### **9.3.5. Class and Instance Variables**

Generally speaking, instance variables are for data unique to each instance and class variables are for attributes and methods shared by all instances of the class:

**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'

As discussed in [A Word About Names and Objects](https://docs.python.org/3/tutorial/classes.html#tut-object), shared data can have possibly surprising effects with involving [mutable](https://docs.python.org/3/glossary.html#term-mutable) objects such as lists and dictionaries. For example, 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']

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

Data attributes may be referenced by methods as well as by ordinary users (“clients”) of an object. In other words, classes are not usable to implement pure abstract data types. In fact, nothing in Python makes it possible to enforce data hiding — it is all based upon convention. (On the other hand, the Python implementation, written in C, can completely hide implementation details and control access to an object if necessary; this can be used by extensions to Python written in C.)

Clients should use data attributes with care — clients may mess up invariants maintained by the methods by stamping on their data attributes. Note that clients may add data attributes of their own to an instance object without affecting the validity of the methods, as long as name conflicts are avoided — again, a naming convention can save a lot of headaches here.

There is no shorthand for referencing data attributes (or other methods!) from within methods. I find that this actually increases the readability of methods: there is no chance of confusing local variables and instance variables when glancing through a method.

Often, the first argument of a method is called self. This is nothing more than a convention: the name self has absolutely no special meaning to Python. Note, however, that by not following the convention your code may be less readable to other Python programmers, and it is also conceivable that a class browser program might be written that relies upon such a convention.

Any function object that is a class attribute defines a method for instances of that class. It is not necessary that the function definition is textually enclosed in the class definition: assigning a function object to a local variable in the class is also ok. For example:

*# Function defined outside the class*

**def** f1(self, x, y):

**return** min(x, x+y)

**class** **C**:

f = f1

**def** g(self):

**return** 'hello world'

h = g

Now f, g and h are all attributes of class C that refer to function objects, and consequently they are all methods of instances of C — h being exactly equivalent to g. Note that this practice usually only serves to confuse the reader of a program.

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)

Methods may reference global names in the same way as ordinary functions. The global scope associated with a method is the module containing its definition. (A class is never used as a global scope.) While one rarely encounters a good reason for using global data in a method, there are many legitimate uses of the global scope: for one thing, functions and modules imported into the global scope can be used by methods, as well as functions and classes defined in it. Usually, the class containing the method is itself defined in this global scope, and in the next section we’ll find some good reasons why a method would want to reference its own class.

Each value is an object, and therefore has a class (also called its type). It is stored as object.\_\_class\_\_.

## 9.5. Inheritance

Of course, a language feature would not be worthy of the name “class” without supporting inheritance. The syntax for a derived class definition looks like this:

**class** **DerivedClassName**(BaseClassName):

<statement-1>

.

.

.

<statement-N>

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:

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

Execution of a derived class definition proceeds the same as for a base class. When the class object is constructed, the base class is remembered. This is used for resolving attribute references: if a requested attribute is not found in the class, the search proceeds to look in the base class. This rule is applied recursively if the base class itself is derived from some other class.

There’s nothing special about instantiation of derived classes: DerivedClassName() creates a new instance of the class. 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. Because methods have no special privileges when calling other methods of the same object, a method of a base class that calls another method defined in the same base class may end up calling a method of a derived class that overrides it. (For C++ programmers: all methods in Python are effectively virtual.)

An overriding method in a derived class may in fact want to extend rather than simply replace the base class method of the same name. There is a simple way to call the base class method directly: just call BaseClassName.methodname(self, arguments). This is occasionally useful to clients as well. (Note that this only works if the base class is accessible as BaseClassName in the global scope.)

Python has two built-in functions that work with inheritance:

* Use [isinstance()](https://docs.python.org/3/library/functions.html" \l "isinstance" \o "isinstance) to check an instance’s type: isinstance(obj, int) will be True only if obj.\_\_class\_\_ is [int](https://docs.python.org/3/library/functions.html#int) or some class derived from [int](https://docs.python.org/3/library/functions.html#int).
* Use [issubclass()](https://docs.python.org/3/library/functions.html" \l "issubclass" \o "issubclass) to check class inheritance: issubclass(bool, int) is True since [bool](https://docs.python.org/3/library/functions.html#bool) is a subclass of [int](https://docs.python.org/3/library/functions.html#int). However, issubclass(float, int) is False since [float](https://docs.python.org/3/library/functions.html#float) is not a subclass of [int](https://docs.python.org/3/library/functions.html#int).

### **9.5.1. Multiple Inheritance**

Python supports a form of multiple inheritance as well. A class definition with multiple base classes looks like this:

**class** **DerivedClassName**(Base1, Base2, Base3):

<statement-1>

.

.

.

<statement-N>

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.

In fact, it is slightly more complex than that; the method resolution order changes dynamically to support cooperative calls to [super()](https://docs.python.org/3/library/functions.html#super). This approach is known in some other multiple-inheritance languages as call-next-method and is more powerful than the super call found in single-inheritance languages.

Dynamic ordering is necessary because all cases of multiple inheritance exhibit one or more diamond relationships (where at least one of the parent classes can be accessed through multiple paths from the bottommost class). For example, all classes inherit from [object](https://docs.python.org/3/library/functions.html#object), so any case of multiple inheritance provides more than one path to reach [object](https://docs.python.org/3/library/functions.html#object). To keep the base classes from being accessed more than once, the dynamic algorithm linearizes the search order in a way that preserves the left-to-right ordering specified in each class, that calls each parent only once, and that is monotonic (meaning that a class can be subclassed without affecting the precedence order of its parents). Taken together, these properties make it possible to design reliable and extensible classes with multiple inheritance. For more detail, see <https://www.python.org/download/releases/2.3/mro/>.

## 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 (whether it is a function, a method or a data member). It should be considered an implementation detail and subject to change without notice.

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. This mangling is done without regard to the syntactic position of the identifier, as long as it occurs within the definition of a class.

Name mangling is helpful for letting subclasses override methods without breaking intraclass method calls. For example:

**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)

The above example would work even if MappingSubclass were to introduce a \_\_update identifier since it is replaced with \_Mapping\_\_update in the Mapping class and \_MappingSubclass\_\_update in the MappingSubclass class respectively.

Note that the mangling rules are designed mostly to avoid accidents; it still is possible to access or modify a variable that is considered private. This can even be useful in special circumstances, such as in the debugger.

Notice that code passed to exec() or eval() does not consider the classname of the invoking class to be the current class; this is similar to the effect of the global statement, the effect of which is likewise restricted to code that is byte-compiled together. The same restriction applies to getattr(), setattr() and delattr(), as well as when referencing \_\_dict\_\_ directly.

## 9.7. Odds and Ends

Sometimes it is useful to have a data type similar to the Pascal “record” or C “struct”, bundling together a few named data items. An empty class definition will do nicely:

**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

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.

Instance method objects have attributes, too: m.\_\_self\_\_ is the instance object with the method m(), and m.\_\_func\_\_ is the function object corresponding to the method.

## 9.8. Iterators

By now you have probably noticed that most container objects can be looped over using a [for](https://docs.python.org/3/reference/compound_stmts.html#for) statement:

**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='')

This style of access is clear, concise, and convenient. The use of iterators pervades and unifies Python. Behind the scenes, the [for](https://docs.python.org/3/reference/compound_stmts.html#for) statement calls [iter()](https://docs.python.org/3/library/functions.html" \l "iter" \o "iter) on the container object. The function returns an iterator object that defines the method [\_\_next\_\_()](https://docs.python.org/3/library/stdtypes.html#iterator.__next__) which accesses elements in the container one at a time. When there are no more elements, [\_\_next\_\_()](https://docs.python.org/3/library/stdtypes.html#iterator.__next__) raises a [StopIteration](https://docs.python.org/3/library/exceptions.html" \l "StopIteration" \o "StopIteration) exception which tells the for loop to terminate. You can call the [\_\_next\_\_()](https://docs.python.org/3/library/stdtypes.html#iterator.__next__) method using the [next()](https://docs.python.org/3/library/functions.html#next) built-in function; this example shows how it all works:

>>>

**>>>** 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

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\_\_()](https://docs.python.org/3/library/stdtypes.html#iterator.__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](https://docs.python.org/3/glossary.html#term-generator) are a simple and powerful tool for creating iterators. They are written like regular functions but use the [yield](https://docs.python.org/3/reference/simple_stmts.html#yield) statement whenever they want to return data. Each time [next()](https://docs.python.org/3/library/functions.html#next) is called on it, the generator resumes where it left off (it remembers all the data values and which statement was last executed). An example shows that generators can be trivially easy to create:

**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

Anything that can be done with generators can also be done with class-based iterators as described in the previous section. What makes generators so compact is that the \_\_iter\_\_() and [\_\_next\_\_()](https://docs.python.org/3/reference/expressions.html#generator.__next__) methods are created automatically.

Another key feature is that the local variables and execution state are automatically saved between calls. This made the function easier to write and much more clear than an approach using instance variables like self.index and self.data.

In addition to automatic method creation and saving program state, when generators terminate, they automatically raise [StopIteration](https://docs.python.org/3/library/exceptions.html" \l "StopIteration" \o "StopIteration). In combination, these features make it easy to create iterators with no more effort than writing a regular function.

## 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.

Examples:

>>>

**>>>** 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']

**Footnotes**

[1](https://docs.python.org/3/tutorial/classes.html#id1)

Except for one thing. Module objects have a secret read-only attribute called [\_\_dict\_\_](https://docs.python.org/3/library/stdtypes.html#object.__dict__) which returns the dictionary used to implement the module’s namespace; the name [\_\_dict\_\_](https://docs.python.org/3/library/stdtypes.html#object.__dict__) is an attribute but not a global name. Obviously, using this violates the abstraction of namespace implementation, and should be restricted to things like post-mortem debuggers.

##############################

8. Errors and Exceptions

Until now error messages haven’t been more than mentioned, but if you have tried out the examples you have probably seen some. There are (at least) two distinguishable kinds of errors: *syntax errors* and *exceptions*.

8.1. Syntax Errors

Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python:

>>>

**>>> while** **True** print('Hello world')

File "<stdin>", line 1

**while** **True** print('Hello world')

^

SyntaxError: invalid syntax

The parser repeats the offending line and displays a little ‘arrow’ pointing at the earliest point in the line where the error was detected. The error is caused by (or at least detected at) the token *preceding* the arrow: in the example, the error is detected at the function [print()](https://docs.python.org/3/library/functions.html#print), since a colon (':') is missing before it. File name and line number are printed so you know where to look in case the input came from a script.

8.2. Exceptions

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called *exceptions* and are not unconditionally fatal: you will soon learn how to handle them in Python programs. Most exceptions are not handled by programs, however, and result in error messages as shown here:

>>>

**>>>** 10 \* (1/0)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

ZeroDivisionError: division by zero

**>>>** 4 + spam\*3

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

NameError: name 'spam' is not defined

**>>>** '2' + 2

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: can only concatenate str (not "int") to str

The last line of the error message indicates what happened. Exceptions come in different types, and the type is printed as part of the message: the types in the example are [ZeroDivisionError](https://docs.python.org/3/library/exceptions.html" \l "ZeroDivisionError" \o "ZeroDivisionError), [NameError](https://docs.python.org/3/library/exceptions.html" \l "NameError" \o "NameError) and [TypeError](https://docs.python.org/3/library/exceptions.html" \l "TypeError" \o "TypeError). The string printed as the exception type is the name of the built-in exception that occurred. This is true for all built-in exceptions, but need not be true for user-defined exceptions (although it is a useful convention). Standard exception names are built-in identifiers (not reserved keywords).

The rest of the line provides detail based on the type of exception and what caused it.

The preceding part of the error message shows the context where the exception occurred, in the form of a stack traceback. In general it contains a stack traceback listing source lines; however, it will not display lines read from standard input.

[Built-in Exceptions](https://docs.python.org/3/library/exceptions.html#bltin-exceptions) lists the built-in exceptions and their meanings.

8.3. Handling Exceptions

It is possible to write programs that handle selected exceptions. Look at the following example, which asks the user for input until a valid integer has been entered, but allows the user to interrupt the program (using Control-C or whatever the operating system supports); note that a user-generated interruption is signalled by raising the [KeyboardInterrupt](https://docs.python.org/3/library/exceptions.html" \l "KeyboardInterrupt" \o "KeyboardInterrupt) exception.

>>>

**>>> while** **True**:

**...**  **try**:

**...**  x = int(input("Please enter a number: "))

**...**  **break**

**...**  **except** **ValueError**:

**...**  print("Oops! That was no valid number. Try again...")

**...**

The [try](https://docs.python.org/3/reference/compound_stmts.html#try) statement works as follows.

* First, the *try clause* (the statement(s) between the [try](https://docs.python.org/3/reference/compound_stmts.html#try) and [except](https://docs.python.org/3/reference/compound_stmts.html#except) keywords) is executed.
* If no exception occurs, the *except clause* is skipped and execution of the [try](https://docs.python.org/3/reference/compound_stmts.html#try) statement is finished.
* If an exception occurs during execution of the [try](https://docs.python.org/3/reference/compound_stmts.html#try) clause, the rest of the clause is skipped. Then, if its type matches the exception named after the [except](https://docs.python.org/3/reference/compound_stmts.html#except) keyword, the *except clause* is executed, and then execution continues after the try/except block.
* If an exception occurs which does not match the exception named in the *except clause*, it is passed on to outer [try](https://docs.python.org/3/reference/compound_stmts.html#try) statements; if no handler is found, it is an *unhandled exception* and execution stops with a message as shown above.

A [try](https://docs.python.org/3/reference/compound_stmts.html#try) statement may have more than one *except clause*, to specify handlers for different exceptions. At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding *try clause*, not in other handlers of the same try statement. An *except clause* may name multiple exceptions as a parenthesized tuple, for example:

... **except** (**RuntimeError**, **TypeError**, **NameError**):

... **pass**

A class in an [except](https://docs.python.org/3/reference/compound_stmts.html#except) clause is compatible with an exception if it is the same class or a base class thereof (but not the other way around — an *except clause* listing a derived class is not compatible with a base class). For example, the following code will print B, C, D in that order:

**class** **B**(**Exception**):

**pass**

**class** **C**(B):

**pass**

**class** **D**(C):

**pass**

**for** cls **in** [B, C, D]:

**try**:

**raise** cls()

**except** D:

print("D")

**except** C:

print("C")

**except** B:

print("B")

Note that if the *except clauses* were reversed (with except B first), it would have printed B, B, B — the first matching *except clause* is triggered.

All exceptions inherit from [BaseException](https://docs.python.org/3/library/exceptions.html" \l "BaseException" \o "BaseException), and so it can be used to serve as a wildcard. Use this with extreme caution, since it is easy to mask a real programming error in this way! It can also be used to print an error message and then re-raise the exception (allowing a caller to handle the exception as well):

**import** **sys**

**try**:

f = open('myfile.txt')

s = f.readline()

i = int(s.strip())

**except** **OSError** **as** err:

print("OS error: **{0}**".format(err))

**except** **ValueError**:

print("Could not convert data to an integer.")

**except** **BaseException** **as** err:

print(f"Unexpected **{**err**=}**, **{**type(err)**=}**")

**raise**

Alternatively the last except clause may omit the exception name(s), however the exception value must then be retrieved from sys.exc\_info()[1].

The [try](https://docs.python.org/3/reference/compound_stmts.html#try) … [except](https://docs.python.org/3/reference/compound_stmts.html#except) statement has an optional *else clause*, which, when present, must follow all *except clauses*. It is useful for code that must be executed if the *try clause* does not raise an exception. For example:

**for** arg **in** sys.argv[1:]:

**try**:

f = open(arg, 'r')

**except** **OSError**:

print('cannot open', arg)

**else**:

print(arg, 'has', len(f.readlines()), 'lines')

f.close()

The use of the else clause is better than adding additional code to the [try](https://docs.python.org/3/reference/compound_stmts.html#try) clause because it avoids accidentally catching an exception that wasn’t raised by the code being protected by the try … except statement.

When an exception occurs, it may have an associated value, also known as the exception’s *argument*. The presence and type of the argument depend on the exception type.

The *except clause* may specify a variable after the exception name. The variable is bound to an exception instance with the arguments stored in instance.args. For convenience, the exception instance defines \_\_str\_\_() so the arguments can be printed directly without having to reference .args. One may also instantiate an exception first before raising it and add any attributes to it as desired.

>>>

**>>> try**:

**...**  **raise** **Exception**('spam', 'eggs')

**... except** **Exception** **as** inst:

**...**  print(type(inst)) *# the exception instance*

**...**  print(inst.args) *# arguments stored in .args*

**...**  print(inst) *# \_\_str\_\_ allows args to be printed directly,*

**...**  *# but may be overridden in exception subclasses*

**...**  x, y = inst.args *# unpack args*

**...**  print('x =', x)

**...**  print('y =', y)

**...**

<class 'Exception'>

('spam', 'eggs')

('spam', 'eggs')

x = spam

y = eggs

If an exception has arguments, they are printed as the last part (‘detail’) of the message for unhandled exceptions.

Exception handlers don’t just handle exceptions if they occur immediately in the *try clause*, but also if they occur inside functions that are called (even indirectly) in the *try clause*. For example:

>>>

**>>> def** this\_fails():

**...**  x = 1/0

**...**

**>>> try**:

**...**  this\_fails()

**... except** **ZeroDivisionError** **as** err:

**...**  print('Handling run-time error:', err)

**...**

Handling run-time error: division by zero

8.4. Raising Exceptions

The [raise](https://docs.python.org/3/reference/simple_stmts.html#raise) statement allows the programmer to force a specified exception to occur. For example:

>>>

**>>> raise** **NameError**('HiThere')

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

NameError: HiThere

The sole argument to [raise](https://docs.python.org/3/reference/simple_stmts.html#raise) indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from [Exception](https://docs.python.org/3/library/exceptions.html#Exception)). If an exception class is passed, it will be implicitly instantiated by calling its constructor with no arguments:

**raise** **ValueError** *# shorthand for 'raise ValueError()'*

If you need to determine whether an exception was raised but don’t intend to handle it, a simpler form of the [raise](https://docs.python.org/3/reference/simple_stmts.html#raise) statement allows you to re-raise the exception:

>>>

**>>> try**:

**...**  **raise** **NameError**('HiThere')

**... except** **NameError**:

**...**  print('An exception flew by!')

**...**  **raise**

**...**

An exception flew by!

Traceback (most recent call last):

File "<stdin>", line 2, in <module>

NameError: HiThere

8.5. Exception Chaining

The [raise](https://docs.python.org/3/reference/simple_stmts.html#raise) statement allows an optional [from](https://docs.python.org/3/reference/simple_stmts.html#raise) which enables chaining exceptions. For example:

*# exc must be exception instance or None.*

**raise** **RuntimeError** **from** **exc**

This can be useful when you are transforming exceptions. For example:

>>>

**>>> def** func():

**...**  **raise** **ConnectionError**

**...**

**>>> try**:

**...**  func()

**... except** **ConnectionError** **as** exc:

**...**  **raise** **RuntimeError**('Failed to open database') **from** **exc**

**...**

Traceback (most recent call last):

File "<stdin>", line 2, in <module>

File "<stdin>", line 2, in func

ConnectionError

The above exception was the direct cause of the following exception:

Traceback (most recent call last):

File "<stdin>", line 4, in <module>

RuntimeError: Failed to open database

Exception chaining happens automatically when an exception is raised inside an [except](https://docs.python.org/3/reference/compound_stmts.html#except) or [finally](https://docs.python.org/3/reference/compound_stmts.html#finally) section. This can be disabled by using from None idiom:

>>>

**>>> try**:

**...**  open('database.sqlite')

**... except** **OSError**:

**...**  **raise** **RuntimeError** **from** None

**...**

Traceback (most recent call last):

File "<stdin>", line 4, in <module>

RuntimeError

For more information about chaining mechanics, see [Built-in Exceptions](https://docs.python.org/3/library/exceptions.html#bltin-exceptions).

8.6. User-defined Exceptions

Programs may name their own exceptions by creating a new exception class (see [Classes](https://docs.python.org/3/tutorial/classes.html#tut-classes) for more about Python classes). Exceptions should typically be derived from the [Exception](https://docs.python.org/3/library/exceptions.html#Exception) class, either directly or indirectly.

Exception classes can be defined which do anything any other class can do, but are usually kept simple, often only offering a number of attributes that allow information about the error to be extracted by handlers for the exception. When creating a module that can raise several distinct errors, a common practice is to create a base class for exceptions defined by that module, and subclass that to create specific exception classes for different error conditions:

**class** **Error**(**Exception**):

*"""Base class for exceptions in this module."""*

**pass**

**class** **InputError**(Error):

*"""Exception raised for errors in the input.*

*Attributes:*

*expression -- input expression in which the error occurred*

*message -- explanation of the error*

*"""*

**def** \_\_init\_\_(self, expression, message):

self.expression = expression

self.message = message

**class** **TransitionError**(Error):

*"""Raised when an operation attempts a state transition that's not*

*allowed.*

*Attributes:*

*previous -- state at beginning of transition*

*next -- attempted new state*

*message -- explanation of why the specific transition is not allowed*

*"""*

**def** \_\_init\_\_(self, previous, next, message):

self.previous = previous

self.next = next

self.message = message

Most exceptions are defined with names that end in “Error”, similar to the naming of the standard exceptions.

Many standard modules define their own exceptions to report errors that may occur in functions they define. More information on classes is presented in chapter [Classes](https://docs.python.org/3/tutorial/classes.html#tut-classes).

8.7. Defining Clean-up Actions

The [try](https://docs.python.org/3/reference/compound_stmts.html#try) statement has another optional clause which is intended to define clean-up actions that must be executed under all circumstances. For example:

>>>

**>>> try**:

**...**  **raise** **KeyboardInterrupt**

**... finally**:

**...**  print('Goodbye, world!')

**...**

Goodbye, world!

**KeyboardInterrupt**

Traceback (most recent call last):

File "<stdin>", line 2, in <module>

If a [finally](https://docs.python.org/3/reference/compound_stmts.html#finally) clause is present, the finally clause will execute as the last task before the [try](https://docs.python.org/3/reference/compound_stmts.html#try) statement completes. The finally clause runs whether or not the try statement produces an exception. The following points discuss more complex cases when an exception occurs:

* If an exception occurs during execution of the try clause, the exception may be handled by an [except](https://docs.python.org/3/reference/compound_stmts.html#except) clause. If the exception is not handled by an except clause, the exception is re-raised after the finally clause has been executed.
* An exception could occur during execution of an except or else clause. Again, the exception is re-raised after the finally clause has been executed.
* If the finally clause executes a [break](https://docs.python.org/3/reference/simple_stmts.html#break), [continue](https://docs.python.org/3/reference/simple_stmts.html#continue) or [return](https://docs.python.org/3/reference/simple_stmts.html#return) statement, exceptions are not re-raised.
* If the try statement reaches a [break](https://docs.python.org/3/reference/simple_stmts.html#break), [continue](https://docs.python.org/3/reference/simple_stmts.html#continue) or [return](https://docs.python.org/3/reference/simple_stmts.html#return) statement, the finally clause will execute just prior to the break, continue or return statement’s execution.
* If a finally clause includes a return statement, the returned value will be the one from the finally clause’s return statement, not the value from the try clause’s return statement.

For example:

>>>

**>>> def** bool\_return():

**...**  **try**:

**...**  **return** **True**

**...**  **finally**:

**...**  **return** **False**

**...**

**>>>** bool\_return()

False

A more complicated example:

>>>

**>>> def** divide(x, y):

**...**  **try**:

**...**  result = x / y

**...**  **except** **ZeroDivisionError**:

**...**  print("division by zero!")

**...**  **else**:

**...**  print("result is", result)

**...**  **finally**:

**...**  print("executing finally clause")

**...**

**>>>** divide(2, 1)

result is 2.0

executing finally clause

**>>>** divide(2, 0)

division by zero!

executing finally clause

**>>>** divide("2", "1")

executing finally clause

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "<stdin>", line 3, in divide

TypeError: unsupported operand type(s) for /: 'str' and 'str'

As you can see, the [finally](https://docs.python.org/3/reference/compound_stmts.html#finally) clause is executed in any event. The [TypeError](https://docs.python.org/3/library/exceptions.html" \l "TypeError" \o "TypeError) raised by dividing two strings is not handled by the [except](https://docs.python.org/3/reference/compound_stmts.html#except) clause and therefore re-raised after the finally clause has been executed.

In real world applications, the [finally](https://docs.python.org/3/reference/compound_stmts.html#finally) clause is useful for releasing external resources (such as files or network connections), regardless of whether the use of the resource was successful.

8.8. Predefined Clean-up Actions

Some objects define standard clean-up actions to be undertaken when the object is no longer needed, regardless of whether or not the operation using the object succeeded or failed. Look at the following example, which tries to open a file and print its contents to the screen.

**for** line **in** open("myfile.txt"):

print(line, end="")

The problem with this code is that it leaves the file open for an indeterminate amount of time after this part of the code has finished executing. This is not an issue in simple scripts, but can be a problem for larger applications. The [with](https://docs.python.org/3/reference/compound_stmts.html#with) statement allows objects like files to be used in a way that ensures they are always cleaned up promptly and correctly.

**with** open("myfile.txt") **as** f:

**for** line **in** f:

print(line, end="")

After the statement is executed, the file *f* is always closed, even if a problem was encountered while processing the lines. Objects which, like files, provide predefined clean-up actions will indicate this in their documentation.