Chapter 3

Defining Names in and from Modules

A little inaccuracy sometimes saves tons of explanation.
Hector Hugh Munro

When certain concepts of TEX are introduced informally, general rules will be stated; afterwards you will find that the rules aren't strictly true. In general, the later chapters contain more reliable information than the earlier ones do. The author feels that this technique of deliberate lying will actually make it easier for you to learn the ideas. Once you understand a simple but false rule, it will not be hard to supplement that rule with its exceptions.
Donald Knuth

You can't handle the truth.
Colonel Nathan R. Jessep (from the movie "A Few Good Men")

Chapter Objectives

- Learn basic information about the concepts of names, objects, and rules
- Learn about module objects, modules as files, and script modules
- Learn how to define and delete names, redefine names, and share objects
- Learn why Python raises exceptions when it cannot execute a statement
- Learn how to draw module and value objects, and names (in namespaces)
- Learn how to import modules and attribute names from other modules
- Learn how to run the Python interpreter and experiment with Python

3.1 Introduction

Python (the language and system) can be described in terms of three general concepts: names, objects, and rules

Python is both a programming language and a system that runs programs. Understanding Python requires knowing the meanings and interrelationships of just three general concepts: (attribute) names, objects, and rules. The more we use these concepts when learning and discussing Python, and the more we write, analyze, and describe Python code using them, the better we will understand these concepts. For now, we quickly summarize these concepts at a level suitable for use in this chapter; in later chapters, we explore these concepts in more detail. Because these concepts are interrelated, read these descriptions more than once and examine the pictures that illustrate them in Section 3.3.

Names refer/are–bound to objects. We can define a name and bind it to an object, find the object a name refers/is–bound to, change what object a name refers/is–bound to, and delete a name. A name in Python is an attribute in some object's namespace, so we also refer to names as attribute names; in this chapter, the names we define and use are attributes in the namespace of module objects.

Names refer to objects and in this chapter are attributes in the namespace of module objects

Objects represent all aspects of Python programs. There are four major categories of objects: modules, values, functions, and classes. Many objects provide namespaces in which to define (attribute) names. Value objects store special data/values; they are also known as instance objects, because they are instances of a class object that specifies the type of the value/instance object.

Python has four categories for objects, representing modules, values/instances, functions, and classes; they also provide namespaces for attribute names

Rules describe the processes Python uses to execute[1] programs. This chapter focuses on four important rules that operate on namespaces and the bindings between names and objects. These rules specify how to

- define a name in a namespace and make–it–refer/bind–it to an object
- find the object to which a name refers/is–bound
- rebind a name (make it refer) to a different object
- remove a name from a namespace

Rules describe the processes Python uses to execute programs

This chapter explores how to manipulate (attribute) names in the namespaces of module objects. We use the terms name and attribute name interchangeably: every name is an attribute in the namespace of some object. Likewise, we might say a name refers to an object or say a name is bound to an object, or even an object is bound to a name: refers is an asymmetric term: a name refers to an object, not vice–versa; but bounds is symmetric term: names and objects are bound together.

Python uses the terms name and attribute name similarly; likewise we will say either "names refer to objects" or "names are bound to objects"

Finally, the EBNF rules from Chapter 2 were the truth, the whole truth, and nothing but the truth. But, starting in this chapter, we will sometimes present EBNF rules that are correct, but missing some options governing complex parts of Python. We need to learn the simple parts of the language easily, and apply everything we learn concretely; so trying to learn all the details and options now would cause information overload and confusion. Therefore, the EBNF rules from now on will be correct, but not complete: the truth and nothing but the truth; but we will spiral towards the whole truth. The incomplete EBNF rules will use footnotes that briefly discuss their omissions, which later chapters will supply.

This book often uses a spiral approach: the simplest forms of some Python language features are covered first, and then more options and details are gradually exposed

3.2 Modules, Files, and Scripts

Module objects are fundamental in Python, because all Python code is written in files that Python translates into module objects. For example, Python knows that a module named `math` will reside in the file `math.py`, which comprises the module/file name `math` followed by a dot, followed by the file extension `py`, which is Python's standard file extension for modules. Many modules and files use names that are this simple, but module names can be more complex in Python, according to the following EBNF rule.

Python translates code from files into module objects

[1] Python executes a program by executing the instructions/statements that comprise the program. Execute in this context in a synonym of "do", "perform", or "carry out".

EBNF Description: *module_name* (Python module names)
name ⇐ *identifier* (*name* is now a synonym for *identifier*)
qualified_name ⇐ *name*{ . *name*}
module_name ⇐ *qualified_name* (*module_name* is now a synonym for *qualified_name*)

Module names can be simple or qualified: multiple names separated by dots

In qualified module names (multiple names separated by dots) Python uses the last name as the file name of the module; it uses all prior names as the folder/directory path in which to find the file. For the module name `ics31.courselib.prompt` Python will look for the file `prompt.py` inside the folder/directory `courselib`, which itself is inside the folder/directory `ics31`.

We start a task by telling Python which module to execute; that module is known as the script for the task

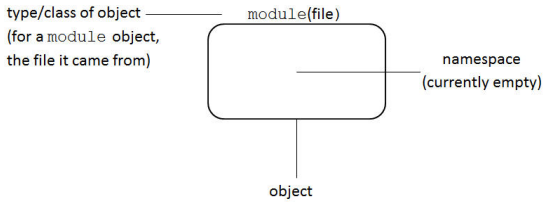
Python programs typically comprise many modules, cooperating to implement some task: some modules work directly to solve the task; other modules (i.e., software components or libraries) are more general and are used in the implementation of many diverse tasks. When we tell Python to execute a task starting with a specific module, that module is called the "script": think of the programmer as a playwright, and Python as the actor. When told to execute a script, Python performs a sequence of four actions. It will

1. find the script's file using the convention just discussed for modules
2. translate all the statements in the script's file into Python's bytecode[2]
3. create a module object for the script, initially with an empty namespace
4. execute its bytecode, updating the namespace of the script module object

We draw all objects as rounded–edge rectangles, labeled on its top by the type of value the object represents

Figure 3.1 illustrates a module object that has an empty namespace. In this book, rounded–edge rectangles always represent objects, with the type/class of the object appearing as a label on top. The object in Figure 3.1 is a module object, whose label also includes the module's file name. We will soon see other objects, with many labeled by one of Python's built in types (e.g., *int*, *bool*, *str*) that we learned in Section 2.5 when studying literals.

Figure 3.1: The Parts of a Module Object with an Empty Namespace



Namespaces of objects store names and storage cells, which refer to objects

Module objects show their namespaces inside their rectangles. This namespace is empty, but in Section 3.3 we will learn how to define names and their values in the namespace of a module object, and how to illustrate such definitions: as names and their storage cells, which store references (represented by arrows) to the objects they refer to (more rounded–edged rectangles). At that

[2]Bytecode is a special language that is lower–level than Python, but is higher–level than machine code. Python translates modules into bytecode and then its virtual machine executes these modules by interpreting their bytecode. This chapter does not explore bytecode further, pretending that Python executes modules directly, without having to translate them into bytecode. But, if we studied bytecode, we would learn that Python stores the bytecode of imported modules in special "cached" files, (with the same module name, but using the extension `.pyc`) in the `__pycache__` folder. When we tell Python to import a module, if it finds that the module's `.py` file remained unchanged after its `.pyc` file was stored, Python immediately uses the `.pyc` bytecode file, avoiding retranslating the Python module into bytecode.

time we will also learn about the special `sys` module, and how it keeps track of the script being run and any other modules that the script imports.

3.3 Defining Names in a Module

Both assignment statements and import statements define names in the namespace of a module object

There are two ways to define names in modules; each adds attribute names to the namespace of a module object and binds these names to other objects. The first is the *assignment_statement* whose EBNF appears below[3] The second is the *import_statement*, which is discussed in Section 3.4 and builds on the material covered here. Note that before the *expression* EBNF rule, we define *literal* as any form of literal from Python's built in types. The *assignment_statement* itself features the `=` delimiter separating the *name* from the *expression*

EBNF Description: *assignment_statement* (Defining a name and binding it to an object)

literal	⇐ *int_literal* | *float_literal* | *imaginary_literal* | *bool_literal* | *str_literal* | *bytes_literal* | *none_literal*
expression	⇐ *literal* | *name*
assignment_statement	⇐ *name* `=` *expression*

Semantics of Assignment Statements

Semantically, Python executes an *assignment_statement* according to the three rules below. In the next section we will learn how to illustrate assignment statements by producing pictures illustrating these semantics.

1. Find the object denoted by *expression* on the right side of the `=` delimiter.
   A. If *expression* is a *literal*, check if a value object for that literal already exists: if so, use it; otherwise, create a new value object storing that literal value and use it
   B. If *expression* is a *name*, check if that name is already in the namespace for the module object; if so, find the object it refers to and use it; otherwise, raise the `NameError` exception, reporting a problem. See Section 3.3.3 for details about exceptions.
2. Find the *name* on the left side of the `=` delimiter. If it is not already in the namespace for the module object, add it to the namespace.
3. Make the *name* in the namespace refer to the object found by *expression*.

Assignment statements in Python have a different meaning from equality statements in Mathematics

Assignment statements in Python seem like statements of equality in Mathematics, but they are different. In mathematics = is symmetric: $x = 1$ and $1 = x$ have the same meaning. Assignment statements in Python are asymmetric: the left side of the `=` delimiter must be a *name* while the right side must be an *expression*, whose EBNF, at present, allows only a *literal* or *name*. In Python, we read the *assignment_statement* `x = 1` as `x` gets (is assigned) `1`. The symbols `1 = x` are illegal in Python because they do not match the EBNF.

We will soon learn about more complicated expressions, which include functions and operators

In the two next chapters the *expression* EBNF rule will be updated and we will learn how to perform complex calculations, using Python's built in functions and operators, So, the `=` here is like the ⇐ symbol in EBNF rules, whose left side is just a rule name, but whose right right side can be a complex description.

A defined name will always refer to some object

Finally, note that once a name is defined, it will always refer to some object; of course, we can always bind a name to the value object representing the literal `None`, from the `None_Type` class; e.g., `x = None`.

[3] Omitted from this EBNF description: using more complex `name` and `expression` rules.

3.3.1 Illustrating Names and their Bindings in Modules

Pictures help us understand the Python and understand the code we write in Python

To help us learn Python, this book will often present pictures that illustrate the concepts it is teaching. This is especially true for understanding the semantics of Python statements, like the *assignment_statement* and the *import_statement* covered in this chapter. Acquiring skill in reading, understanding, and drawing such pictures on our own is an important step in learning general concepts and being able to analyze specific code. It will help us predict what our code will do, and give us an excellent analysis tool to understand our code if it is not doing what we want —a too common occurrence for novice and expert programmers.

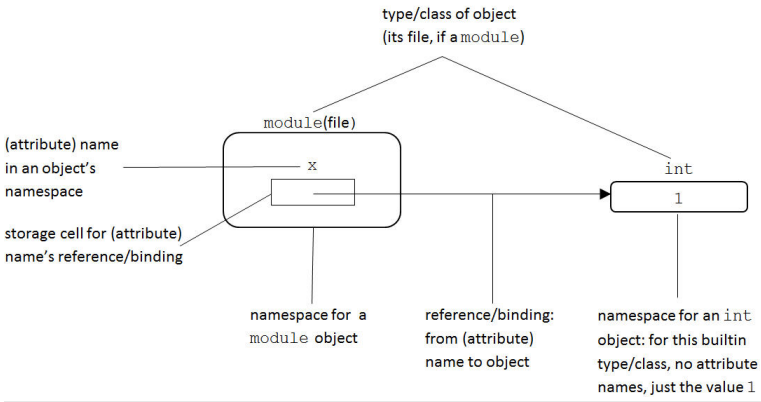
Our need to draw pictures may diminish but it will not disappear

As we become skilled in understanding pictures and become more comfortable with Python, our need for pictures will decrease; but even then, being able to draw pictures when we need them, in a novel, complex, or confusing situation, will be an important skill to keep.

How to draw a name in a namespace and its binding to an object

Figure 3.2 illustrates how to draw module objects, the (attribute) names in their namespaces, and the bindings to the objects these names refer to. In particular, this picture illustrates how Python executes the statement the assignment statement `x = 1` in an unnamed module.

Figure 3.2: Define a Name (in a Namespace) Referring/Bound-to a Value Object



Simple value objects, like those for Python's built in types, store a value but no namespace

Recall that a rounded-edge rectangle always represents an object encapsulating its value and possibly its namespace. Objects of simplest built in types (like the `int` object here) store only a value, and not any attribute names, so we draw them more compactly.

A simple module comprising two assignment statements

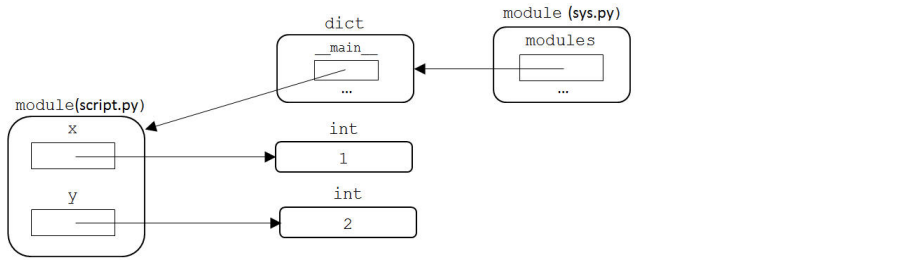
Let us now take a complex (for Chapter 3!) scenario and illustrate it fully. Suppose we direct Python to execute a script from the file named `script.py`, and this file contains the following two assignment statements.

```
1  x = 1
2  y = 2
```

Illustrating how a simple module object executes two assignment statements that update its namespace

Figure 3.3 illustrates not only how we should picture modules, names, objects, and bindings in Python, but also some Python infrastructure: the special module `sys` and the built in type `dict` (dictionary) which implements a non-module namespace.

Figure 3.3: Semantics of Running `script.py` as a Script



At the most basic level, this picture illustrates how we draw module objects, names in their namespaces, and bindings between these names and value objects. Here Python executes the module from the `script.py` file. As described in Section 3.2 Python creates a module object with an empty namespace; it then executes the assignment statements in the the file, updating the namespace according to the semantics described in Section 3.3. Pictorially, each literal is written as a value object, and each newly defined name is added into the namespace of the module's object; each name is made–to–refer/bound–to its value object: the tail of an arrow is placed in the storage cell for the name and the head is pointed to the value object.

In an assignment statement, we add a new name to the namespace of a module object and put an arrow from its storage cell to the object it refers to

This picture also illustrates some other aspects of running Python scripts, which are now briefly described at a level appropriate for this chapter. First, the picture shows a special module from the file `sys.py` with the one attribute name `modules` in its namespace; the `...` at the bottom of this module's object signifies this module defines other names, but they are not of interest to us now. The name `modules` refers to an object of type `dict` which is Python's built in dictionary type, which we will study later in great detail; the purpose of a `dict` object is to act as a namespace.

Some Python infrastructure: how the `sys` module and a `dict` object is updated when Python runs a script

For now, the namespace of the `dict` object contains just one name `__main__` which is the special name that Python binds to the module object that represents the script Python is executing. When we learn how to import modules in Section 3.4, we will see that Python adds the names of those modules to this `dict` and binds them to the imported module objects. Again `...` in the `dict` object signifies at present we are not interested in any other names in this namespace.

When Python runs a script, the `sys.modules` dictionary refers to its module object by the name `__main__`

3.3.2 How Names Share, are Redefined, and are Deleted

This section explores three aspects of names: how two different names can refer to the same object (called "sharing" or "aliasing"); how we can redefine which object a name refers to; and how to remove names from their namespace.

Three interesting observations about names

Sharing Objects: Every name defined in Python refers to one object; but an object may be referred to by many names. Again an interesting asymmetry. We will need to learn more Python before we can explain why such sharing is not only useful, but critical to our ability to write code to implement complex tasks. But we can easily explain how it works now. Figure 3.4 illustrates how we can

Every name refers to one object; but an object can be referred to by multiple names

share/alias an object in our code by executing the simple script below. For the next few illustrations, we will omit the the `sys` and `dict` objects, because we don't want to distract from the primary issues being explore here.

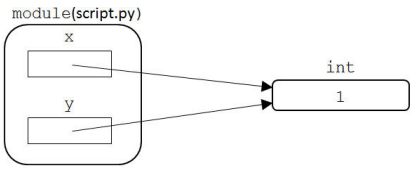
`script` module

```
1  x = 1
2  y = x
```

In sharing, the storage cells for two different names show arrows that point to the same object

Notice that `x` is bound to an object of type `int` and then `y` is bound to the same object `x` is bound to. Rules 1B and 3 for the semantics of the *assignment_statement* say that if the *expression* is a name, then use the object that the name refers to. So the name on the left of the `=` delimiter is bound to same object that is denoted by the *expression* on the right of the `=` delimiter. In the picture, we put into the storage cell for `y` an arrow that refers to the same object that the storage cell for `x` refers to. This is really not a complicated rule, but students have a devil of a time learning it and applying it consistently.

Figure 3.4: Names Sharing/Aliasing an Object



The arrows shown in storage cells always point at objects, never at other storage cells

Confused beginners sometimes show this picture with the arrow in the storage cell for `y` pointing to the storage cell for `x`, instead of pointing to the same object the storage cell for `x` points too. Fundamentally this is wrong because arrows refer to object and the storage cell for `x` is not an objects; remember that all objects appear as rounded-edge rectangles.

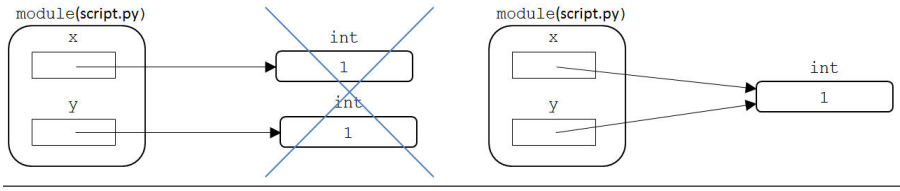
Names share/alias the objects created for literal values, mostly

A much more interesting issue surrounds Rule 1A for the semantics of the *assignment_statement*. It states, "If *expression* is a *literal*, check if a value object for that literal already exists: if so, use it; otherwise, create a new value object storing that literal value and use it." Figure 3.4 illustrates how the only mention of the literal `1` created a new object. But, Figure 3.5 illustrates how one such value object is aliased as a result of executing the simple script below. The picture below on the right is the one that follows the rules: the first *assignment_statement* creates a new value object storing `1`; the second aliases the same object created by the first, as a result of this rule.

`script` module

```
1  x = 1
2  y = 1
```

Figure 3.5: Two Interpretations of a Simple Script: The Right is Correct



But the picture similar to the one on the left would be correct if instead of `1` we assigned both names `123456789101112`. The problem is that the whole truth is surprisingly complex here. For now we will leave this issue underspecified, always using the interpretation on the right until we resolve it later.

Names sometimes do not share/alias the objects created for literal values

Obviously we see two different pictures: can Python tell the difference? Yes it can. Chapter **??** will explain an operator[4] that allows us determine whether `x` and `y` share an object (as on the right) or refer to two different value objects that store the same value (as on the left).

Python can determine whether two names refer to the same or different objects

Redefining: We can redefine a name, making it refer to a different object. In Section 3.3 Rules 2–3 for the semantics of the *assignment_statement* say that if the *name* on the left of the `=` delimiter is already defined in the namespace, make that name refer to the object denoted by *expression*, replacing the arrow in the storage cell for the name by a new arrow pointing to a different object.

We can redefine a name: make a name refer to a different object

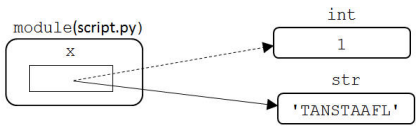
When we redefine a name, its previous reference/arrow is gone and forgotten by Python, but sometimes in such cases —when we want to emphasize the redefinition of a name— instead of erasing the arrow we will illustrate the previous reference in a storage cell by a dashed arrow. Figure 3.6, which uses a dashed arrow, illustrates how Python executes the simple script below.

A storage cell can store only one reference, although we can write dashed arrows in our pictures to indicate a replaced reference

`script` module

```
1  x = 1
2  x = 'TANSTAAFL'
```

Figure 3.6: Redefining a Name



Notice that at first `x` is bound to an object of type `int` but then it is rebound to an object of type `str`. When a name is redefined, it can refer to any type of object, not just the type of the original object it was bound to.

Names can refer to different types of objects

Undefining: We can remove a name from the namespace of a module's object by executing a *delete_statement*, according to the following EBNF. By using the repetition braces we can use one *delete_statement* to specify the deletion of any number of names.

We can delete a name, which removes it from its namespace

EBNF Description: *del_statement* (Removing a name from an object's namespace)
delete_statement ⇐ `del` *name*{`,`*name*}

Semantically, the specified names are removed from the namespace; in our pictures, we erase or cross–out each name and its storage cell from the namespace, and if erasing, erase the arrow showing its binding. If any specified name does not exist in this namespace, Python raises the same `NameError` exception that was briefly discussed in Section 3.3 in regards to the semantics of the `assignment_statement`; exceptions are discussed more fully in Section 3.3.3. There is one more aspect of deletion that is not yet important to understand fully, but this section is a convenient place to broach the subject.

Python raises an exception if a delete statement specifies a name that is not in a namespace

[4]The `is` operator determines whether two references share the same object, while the `==` operator determines whether two references refer to objects that store the same value.

Every object occupies space in the computer's memory: to store its value and/or namespace. If we delete a name that refers to some object, and no other name shares/aliases/refers–to that object,[5] Python recycles that object's storage, reclaiming its memory for possible reuse in the future, when creating/storing new objects Because this mechanism is automatic (a feature not available in all programming languages) Python programmers generally don't worry much about recycling memory. There are a few subtleties to understanding this mechanism, which this book will cover when appropriate.

Objects occupy memory; if we delete the only name that refers to an object, Python recycles the memory that object occupies

3.3.3 Programming Errors and Raising Exceptions

Sometimes when Python executes a program, it discovers it cannot do what the code says to do. In such cases, Python reports the problem by "raising" an exception, which is as if Python raises its hands in a shrug and says, "I cannot do that". We have already seen one example: Python raising the `NameError` exception in the assignment statement, if the `expression` is a *name* that is not defined. When we discuss the divide operator, we will learn that if we try to divide a number by zero, Python raises the `ZeroDivisionError` exception.

Python raises an exception when it tries to execute code that it cannot perform correctly

Section **??** discusses the `try/except` statement in Python. By using this statement, programmers tell Python what code to try executing, and what to do if that code raises an exception. The `except` part of this statement specifies an exception handler; without an exception handler, Python handles raised exceptions by terminating the program and printing the exception's name and a short error message describing it on the user's console. Section 3.5 demonstrates code that raises exceptions and shows exactly how Python reports such errors.

A `try/except` statement tells Python how to handle exceptions; without an exception handler, Python handles an exception by terminating the program and printing the exception name and a related message on the user's console

Exceptions are an important concept in modern programming languages, and we will explore many different aspects of exceptions throughout this book, using a spiral approach: as we learn more about Python in general, we will learn more about using exceptions in Python. For now, we can learn that exceptions are represented by classes. In Section 3.4.2 we will learn that Python imports about four dozen standard exception names from the special `builtins` module. Later we will learn how to handle raised exceptions, raise exceptions when special conditions occur, and define our own exception classes.

Exceptions are an important but many–faceted language feature; as we learn more about Python, we will learn more about using exceptions in Python

Section Review Exercises

1. Draw a picture to illustrate the semantics of executing code in the two `script` modules below. The only difference in the code in these modules is in the last *assignment_statement*, which are mirror images. Remember to write dashed arrows for removed references.

   first `script` module

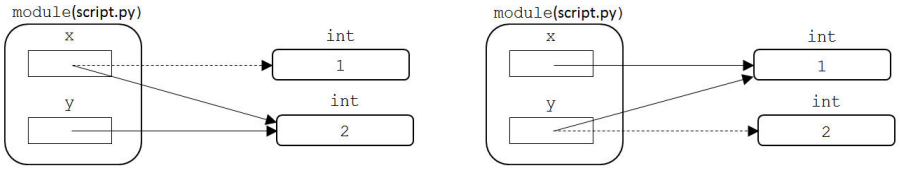
   ```
   1  x = 1
   2  y = 2
   3  x = y
   ```

   second `script` module

   ```
   1  x = 1
   2  y = 2
   3  y = x
   ```

   Answer: On the left, both names point to the bottom object; on the right, both names point to the top object. Remember that assignment statements are asymmetric, so `x = y` has a different meaning than `y = x`.

[5] How does Python know if an object's memory is recyclable? Python stores a reference count for each object, counting how many names refer to it: how much it is shared/aliased. When an object's count goes to 0, Python knows that it can recycle the object's memory.



2. We can write a *name* on each side of the = delimiter in an *assignment_statement*. a. If the name on the left is not already defined in the namespace, what does Python do? b. Same for the name on the right? Hint: asymmetrical.

   ANSWER: a. Python defines the name in the namespace. b. Python raises the `NameError` exception.

3.4 Importing Names of/from Modules

Import statements define names in the namespace of a module object

This section explores a second way to define names in a module: by importing them, using another module. For example, Python's `math` module defines many names: it binds some to value objects (mathematical constants) we can use and others to function objects (mathematical functions) we can call. Specifically, it defines the names `pi` and `e` and binds each to a `float` value approximating the mathematical constants π and e; it defines the names `sqrt`, `sin`, `factorial`, and others, and binds each to a function object that computes that function.

We can import the name of a module, or selected attribute names from a module, or all the attributes names from a module

The EBNF of Python's `import_statement` specifies three different syntactic forms for importing, each having its own semantics. The *import_module* rule imports references to entire module objects; the *import_attribute* rule imports some or all references to objects bound to attribute names in modules.[6] The most important option in each form allows us to define a module or attribute name that is the **same** as the one imported, or to define a *new_name* of our own choosing, which is bound to the same object as the imported name.

EBNF Description: *import_statement* (Importing modules/attributes)

new_name ⇐ *name* (not a *qualified_name*)
module_name_as ⇐ *module_name* [`as` *new_name*]
name_as ⇐ *name* [`as` *new_name*]

import_module ⇐ `import` *module_name_as*{`,`*module_name_as*}
import_attribute ⇐ `from` *module_name* `import` *name_as*{`,`*name_as*} | `from` *module_name* `import *`

import_statement ⇐ *import_module* | *import_attribute*

Semantics Common to all Import Statements

When we refer to a module for the first time (importing its name or an attribute name from it), Python performs the following three actions, and then the semantics of the actual import (which are described afterwards).

1. **Create** a new module object with an empty namespace.
2. **Define** the module's name in the `dict` object and bind it to the created module object. Recall `modules` is a name the the namespace of `sys` module object that refers to the `dict` object, which initially stores the name `__main__` bound to the module object of the running script).

[6] Omitted from this EBNF description: using relative imports from modules in packages.

3. **Execute** all the module's statements, which typically define names in the namespace of the module's object by binding each name to an object.

Python skips the previous steps if the module has already been imported; the `modules` name in the namespace of the `sys` module object stores references to all imported modules

If a program ever imports the same module again, Python skips these three actions, because the module object and all the names in its namespace are already defined; in this case Python immediately performs the semantics specified below for the form of import used. Python determines whether or not a module has already been imported by checking whether or not the `dict` object bound to the `modules` name in the namespace of the special `sys` module object has already defined the name of the imported module.

Semantics for Each Form of Import Statement

The rules specifying the semantics for each form of `import` are explained below, followed by some simple examples of each form, and their meanings. In all cases, Python defines a name: it adds a name to the namespace of the module's object in which the import appears (if it is not already there) and binds that name to the object bound to the imported name.

Import Form 1: `import` *module_name_as*`{,`*module_name_as*`}`
For each *module_name_as* specified in the import, define either *module_name* or *new_name* (if the [`as` *new_name*] option is included) in the importing module and bind it to the object created by the imported the module.

Import Form 2: `from` *module_name* `import` *name_as*`{,`*name_as*`}`
For each *name_as*, specified in the import, define either *name* or *new_name* (if the [`as` *new_name*] option is included) in the importing module and bind it to the same object (sharing the object, as discussed in Section 3.3.2) that *name* is bound to in *module_name*.

Import Form 3: `from` *module_name* `import` *
For each name in the namespace of the *module_name* object, define that name in the importing module and bind it to the same object (sharing the object, as discussed in Section 3.3.2) it is bound to in *module_name*. Exception: if the name starts with an underscore, do not define/bind it in this module; recall that names that start with underscores are special.

All Forms (Raising Exceptions): If Python cannot find the file for an imported module name or the attribute name in an imported module object, it raises the `ImportError` exception, with the message `No module named ...` or `cannot import name ...`

`import_statement`	name(s) to bind in the namespace of the module with the import	object(s) that name(s) is/are bound to
`import math`	`math`	object bound to `math` in `dict` namespace
`import math as m`	`math`	object bound to `math` in `dict` namespace
`import os.path`	`os.path`	object bound to `os.path` in `dict` namespace
`import os.path as op`	`op`	object bound to `os.path` in `dict` namespace
`import math, os.path`	`math/os.path`	objects bound to `math/os.path` in `dict` namespace
`from math import pi`	`pi`	object bound to `pi` in namespace of `math` module
`from math import e as mathe`	`mathe`	object bound to `e` in namespace of `math` module
`from math import *`	`pi`, `e` and all other names defined in the namespace of the `math` module	objects bound to `pi`, `e` and all other names in the namespace of the `math` module

A detailed example illustrating the first two import forms

Figure 3.7 illustrates Python executing the `script` module, which refers to the `imports` module. The four import statements define six names in the `script` object module, some with names from `imports` and some with different names.

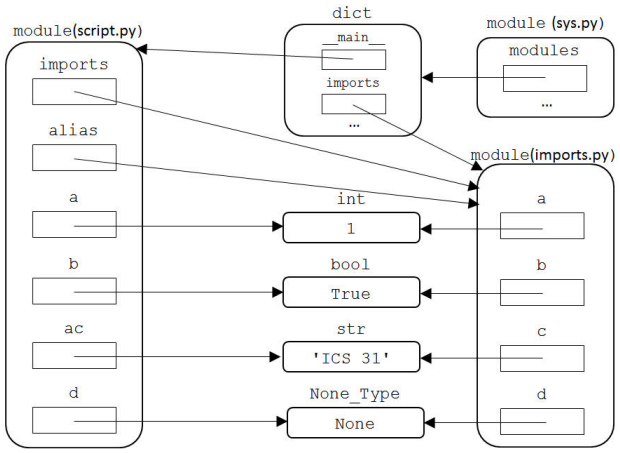
`script` module

```
1 import imports, imports as alias
2 from imports  import a
3 from alias    import b
4 from imports import c as ac, d
```

`imports` module

```
1 a = 1
2 b = True
3 c = 'ICS 31'
4 d = None
```

Figure 3.7: Semantics of Running `script` as a Script, Importing `imports`



Pragmatics of Import Statements

It is a bit premature to discuss the pragmatics of importing. Generally, though, imports all appear at the top of a module (although Python allows them to be intermingled with other statements). Also, there is a preference to not pollute the namespace of the importing module: so using Form 1 (defining just module names) is preferred to Form 2 (defining selected attribute names), which is preferred to Form 3 (defining attribute names for every name from a namespace); but there are circumstances where each form is appropriate.

Recursive imports are possible, and work according to the rules, but they are rarely needed: not at all in this book

Finally, Python allows recursive imports, both direct (a module importing itself) and mutual (two modules each importing the other). While the import rules we have learned specify exactly what happens in these cases, they are complex and tricky to apply, so we will not show any examples here. In fact, recursive imports are rarely needed, and none are used in this book.

3.4.1 Assignment/Delete Statements with Imports

Assignment statements and delete statements can be extended to use the qualified names created by imports

Now that we know how to import modules, we will explore how to use a *qualified_name* (written module name dot attribute name) to manipulate names in the namespace of an imported module: how to find the objects they refer to, redefine their bindings, and undefine them. First, we extend three previously defined EBNF rules to use *qualified_name* instead of *name*.

EBNF Description: Extending*: expression, assignment_statement,* and *delete_statement*

expression ⇐ *literal* | *qualified_name*
assignment_statement ⇐ *qualified_name* `=` *expression*
delete_statement ⇐ `del` *qualified_name*{`,` *qualified_name*}

We can use qualified names to access, redefine, and delete names in modules

So, for example, if a script module includes `import m` and `m` refers to a module object with the attribute name `n` in its namespace, we can write `m.n` in the script to refer to that *qualified_name* name. Here are three ways to use it.

- We can write `m.n` as an *expression* in an *assignment_statement*: `x = m.n` binds `x` to the same object that `m.n` refers to.
- We can write `m.n` as a *qualified_name* on the left side of an *assignment_statement*: `m.n = 1` rebinds `n` in the namespace of the module object `m` to a value object storing the value `1`.
- We can write `m.n` as a *qualified_name* in a *delete_statement*: `del m.n` deletes name `n` from the namespace of module object `m`. We illustrate this deletion by erasing, or placing an X over, the deleted name and its storage cell and the arrow in it.

Extended assignment and delete statements can raise exceptions

For any *qualified_name* used as an *expression* in an `assignment_statement` or specified in the *delete_statement*, the module name must exist and the name must be in the namespace of that module's object; if Python cannot find the attribute name, it raises the `NameError` exception. If a *qualified_name* appears on the left side of the `=` delimiter in an `assignment_statement`, the module name must exist; but if the attribute name doesn't, Python defines that name in the namespace of the module object, as part of the semantics of assignment statements.

One module can manipulate the namespace of another

Figure 3.8, illustrates how Python executes the script below, which imports and manipulates the namespace of the `second` module.

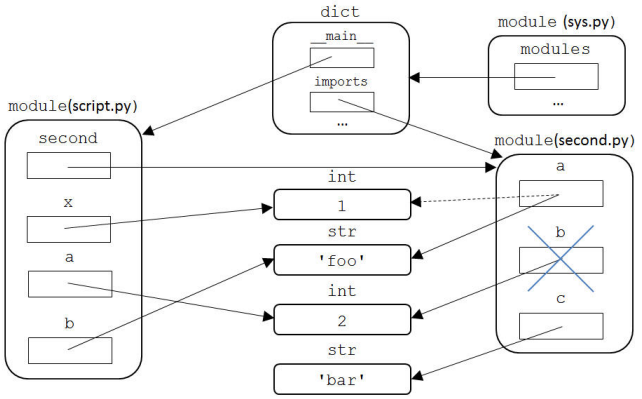
`script` module

```
1 import second                          # import module
2 x = second.a
3 second.a = 'foo'
4 second.c = 'bar'
5 from second import a as b, b as a #import attributes
6 del second.b
```

`second` module

```
1 a = 1
2 b = 2
```

Figure 3.8: Semantics of Running `script` as a Script, importing `second`



Generally, it is bad style for one module to manipulate the name space of another module object

A module normally manipulates names only in its own namespace. Although Python allows one module to manipulate the namespace of another, beginners

should avoid using this feature: it is not a good idea for one module to define, redefine, or delete an attribute name in another. For example, if a script imported the `math` module and executed the *assignment_statement* `math.pi = 3` then for the rest of the program, every access to `math.pi` would result in a reference to a value object with an incorrect value of `3`. Finally, although the module object created from `math.py` is changed, this file remains unchanged; so a different program that that imports the `math` module will create a module object that binds the correct value object to `math.py`.

3.4.2 The `builtins` Module

Python automatically imports the `builtins` module into every imported module

The Python system includes a special module named `builtins`, which Python automatically imports into every Python module, including scripts, as if by the `from builtins import *` statement. By doing so, Python creates a `builtins` module object, defines in its namespace all the attribute names defined in the `builtins.py` file, and imports all these names into the namespace of the script's module object, where we can then easily refer to and use them.

The `builtins` module binds it attribute names to class objects, value objects, function objects

The `builtins` module defines about 100 names. It defines about two dozen names for types and binds them to their class objects, including the names of the seven built in types that we studied when we introduced literals, the name `dict`, and the names of many other important types/classes that we will study and use later in this book. It also defines nine name, including `False`, `None`, and `True` and binds them to their value objects. It defines about four dozen names for exceptions (including `NameError`, `ImportError` and `ZeroDivisionError`) and binds them to them to their class objects. And finally, it defines about three dozen names for functions and binds them to their function objects.

The `builtins` module provides a rich environment for writing code

This module defines names for Python's most basic and important objects, which provide a rich environment for writing code. Chapters **??** and **??** will discuss the meaning and usage of many of the names defined in Python's `builtins` module.

Section Review Exercises

1. Draw a picture to illustrate the semantics of executing code in the `script` module below. Remember to write dashed arrows for removed references.

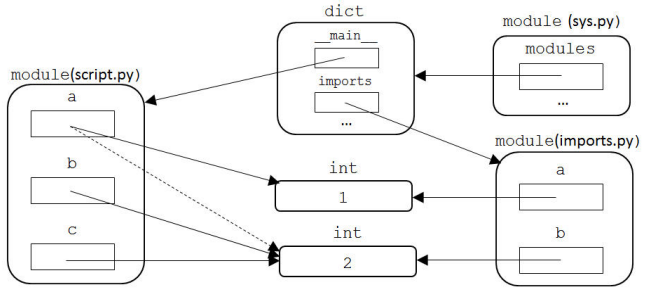
`script` module

```
1 from imports import b as a, b as c
2 from imports import *
```

`imports` module

```
1 a = 1
2 b = 2
```

Answer:



2. Draw a picture to illustrate the semantics of executing the `script` module

below.

`script` module

```
1 import m2,m3
2 x = m2.x
3 y = m3.y
4 from m2 import next.y as z
5 m2.z = 'scriptz'
6 m2.next.x = 1
```

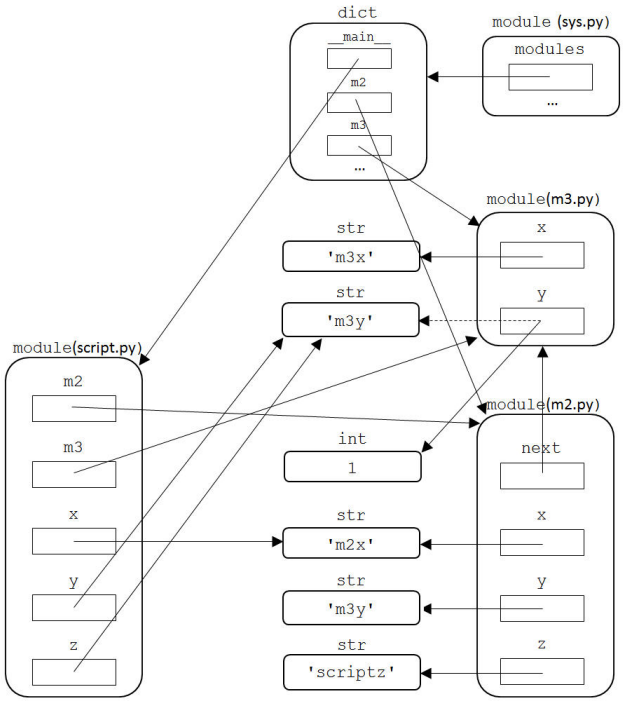
`m2` module

```
1 import m3 as next
2 x = 'm2x'
3 y = 'm2y'
```

`m3` module

```
1 x = 'm3x'
2 y = 'm3y'
```

ANSWER:



3. If the `imports` module defines `_a` (a name starting with an underscore), and the `script` module includes the `from imports import _a` statement, will the name `_a` be imported or not? Justify your answer.

   ANSWER: It will be imported. Names starting with underscores are imported in this form of import, but are not imported in the last form of import, using `*` to import all names not starting with an underscore.

4. In the `script` module below, what value is bound to `pi`? `math.pi`?

   `script` module

   ```
   1 import math
   2 from math import pi
   3 math.pi = 3
   ```

   ANSWER: `pi` is bound to a value object storing π; `math.pi` is bound to a value object storing `3`.

3.5 The Python Interpreter: Experiment!

The Python interpreter is a great resource for exploring Python and discovering its rules of operation

We ave now learned some fundamentally import information about Python modules and their namespaces. We have examined and practiced drawing pictures that illustrate the result of executing Python assignment, delete, and import statements. In the next few chapters, we will learn enough about Python to begin writing and understanding real programs that implement useful tasks. In this section, we will learn the basics of the Python interpreter, which allows us to easily test and validate/correct our knowledge of the Python language features we are learning. The interpreter is a great resource for exploring Python and discovering its rules of operation.

The Python interpreter embodies a Read–Execute–Print loop

The Python interpreter embodies a "Read–Execute–Print" loop. It repeatedly prompts us to enter a Python statement, reads the statement, executes that statement, and finally prints the value produced by executing the statement. The interpreter prompts us to enter statements with the triple chevron: `>>>`. It has a few simple rules/conventions for entering statements.

- We can type a new statement or use the ↑ and ↓ keys to scroll through and select any previously entered statement and edit it by using the ←, →, Del, and Backspace keys (as well as typing, marking, cutting, and pasting). Note that when we enter a statement that is just a literal or name: the Python interpreter executes it by printing the value of the literal or the value bound to the name.
- We can use a single underscore in a statement to refer to the value the interpreter last printed. Note that the assignment, delete, and import statements produce the value `None` when Python executes them: the purpose of these statements is not to produce a value, but instead to change the namespace. The interpreter treats `None` specially: it does not print `None` and does not remember it for future reference by the underscore.
- When we have typed/edited the statement we want, we press ↩ (Enter) to instruct the Python interpreter to read and execute the statement and print its resulting value; if we press ↩ but the statement is incomplete, Python prompts with `...` for the completion. The statement we enter also becomes the new last statement executed, which can be scrolled through; if we mistype a statement, we can easily scroll back to it, edit it, and execute the corrected statement. Remember that if the value produced is not `None` it is printed and becomes the last value printed, which can be referred to via the underscore convention.

In the Python interpreter, the statements we type are the script

When we start the interpreter, it is as if we are entering/executing statements inside the script: the __main__ module. We can import other modules and use the names in their namespaces, but to do so those modules must already exist in files. For now, we will import only special modules that are part of the Python system; but if you know/learn how to write your own files, you can name them with the `py` extension and then import them in the interpreter.

Running the interpreter is an excellent way to explore Python; here are some examples

Figure 3.9 shows an annotated interaction with the Python interpreter for some of the language features in Python that we already know and some simple–to–understand features we will explore soon. Notice how the interpreter reports raised exceptions when incorrect statements are entered.

Figure 3.9: Interpreter Interaction (with annotations)

Entered Statement	Annotation
`>>> 1` `1`	Enter a literal The interpreter prints the value of the literal
`>>> x = _`	Bind `x` to `1` (the value last printed): produce non-printing `None`
`>>> x` `1`	Enter a name: `x` The interpreter prints the value bound to `x`
`>>> import math`	Import the `math` module: produce non-printing `None`
`>>> math.pi` `3.141592653589793`	Enter a qualified name: `pi` from the `math` module The interpreter prints the value bound to `math.pi`
`>>> x = \` `... 2`	Enter part of a statement then the line join character Finish the statement; same as typing `x = 2` on one line
`>>> del x`	Remove `x` from the script's namespace: produce non-printing `None`

`>>> x`
`Traceback (most recent call last):`
`  File "<stdin>", line 1, in <module>`
`NameError: name 'x' is not defined`
	Enter the name `x` (which is no longer in the script's namespace)
	Traceback identifies where error occurred
	In the interpreter, this is always `<stdin>` on line `1`
	Raised exception is `NameError`, with a brief explanation

`>>> 1 = x`
`Traceback (most recent call last):`
`  File "<stdin>", line 1, in <module>`
`SyntaxError: can't assign to literal`
	Bad syntax for an assignment statement
	Traceback identifies where error occurred
	In the interpreter, this is always `<stdin>` on line `1`
	Raised exception is `SyntaxError`, with a brief explanation

`>>> from math import pie`
`Traceback (most recent call last):`
`  File "<stdin>", line 1, in <module>`
`ImportError: cannot import name pie`
	There is no `pie` attribute in the `math` module
	Traceback identifies where error occurred
	In the interpreter, this is always `<stdin>` on line 1
	Raised exception is `ImportError`, with a brief explanation

`>>> math.factorial(5)` `120`	Compute 5!: call the `factorial` function defined in the `math` module The interpreter prints the value computed by this function
`>>> 1*2*3*4*5` `120`	Compute 5!: explicitly use the * (multiply) operators The interpreter prints the value computed by these operators

We can perform experiments with the Python interpreter to verify our understanding or correct our misconceptions

Learning Python requires mastering a lot of information about its syntax and semantics. It takes time to fully understand the rules describing each language feature. How do we know when we understand? We can try experiments with the interpreter: predicting the results of executing statements based on our understanding of the Python's rules, and then running the Python interpreter on those statements. The results produced by the interpreter will either validate our prediction and understanding of the rules, or alert us that our knowledge is deficient, so so we can correct our misunderstanding of the rules. Actively testing our understanding of Python, trying to probe our knowledge to find misconceptions, is an excellent way to improve our understanding.

Chapter Summary

This chapter examined the meaning of names and modules and how they are used in Python. It first discussed the general concepts of names, the objects they refer to and whose namespaces they are part of, and the rules that govern the use of names and objects. We learned that Python translates files into module objects, mostly by defining names; every task that Python implements starts with the execution of a module that is the script for that task. We

discussed assignment statements, which define names in the namespace of a a module object and bind them to a value objects. This discussion included how names can share objects, be redefined to refer to different objects, and be undefined using the delete statement: removed from the namespace of a module object. Throughout this discussion we used pictures to illustrate and help us understand and disambiguate the meanings of these operations. We also learned that sometimes these operations fail, which Python recognizes and reports by raising exceptions. We then explored three different forms of the import statement, each of which allows a module to define names that are bound to value objects that are defined in other modules. Given this capability, and the use of qualified names to specify a name in the namespace of another module object, we re–examined the assignment and delete statements, and briefly discussed the special `builtins` module that is automatically imported into every Python module. Finally, we discussed the Python interpreter and how to enter and execute statements into it; we saw many examples of the statements taught in this chapter, including incorrect statements that raise exceptions. The interpreter is a powerful testbed in which we can explore Python and our understanding of its rules.

Chapter Exercises

1. Assume modules `m1` and `m2` each define an attribute named `n`. a. Describe what problem arises if we write a script that starts with the following two import statements: `from m1 import n` and `from m2 import n` b. Write at least four different pairs of import statements that avoid this problem; for each, write how to refer to the `n` from each module.

2. Assume module `m` defines the names `a` and `b`. Write a module that swaps the values referred to by these names: `a` will ultimately refer to the original values `b` referred to, and `b` will ultimately refer to the original values `a` referred to. Hint: use some form of import and an extra name.

3. This chapter briefly discussed four different exceptions. Name them and briefly explain when Python raises each.

4. In the semantics of the *assignment_statement* EBNF rule: a. Explain in what rule Python raises an exception if it cannot find a name. b. Explain in what rule Python does not raise an exception if it cannot find a name (and say what Python does).

5. After Python executes the `script` module below, what values are bound to `a` and `b` defined in this module? Justify your answer by explaining the action of each line in the `script` module. It might help to draw and update a pictures of the objects involved. Hint: Reread what happens when an already–imported module is imported.

`script` module

```
1 from imports import *
2 a = 10
3 import imports
4 imports.b = 20
5 from imports import *
```

`imports` module

```
1 a = 1
2 b = 2
```