Chapter 4

Reading Headers and Calling Functions

> *Computer Science is a science of abstraction —creating the right model for a problem and devising the appropriate mechanizable techniques to solve it.*
>
> Al Aho and Jeffrey Ullman

Chapter Objectives

- Learn how to understand functions by their headers and semantics
- Learn how to match arguments to their parameters in function calls
- Learn how to call functions and call methods (and their differences)
- Learn the fundamental equation of object–oriented programming
- Learn functions (e.g., input/output) defined in modules and the `str` class

4.1 Introduction

Functions are the most important language feature in Python

Functions are the most important programming language feature in Python. Of the four types of objects that represent all aspects of Python programs (i.e., modules, values, functions, and classes) all relate to functions: module and class objects define functions in their namespaces, function objects represent functions directly, and value objects often call methods (a kind of function used in object–oriented programming). As with other important features in Python, we will learn about functions using a spiral approach: we will learn some basic material about calling functions now, and as we learn more about Python in general, we will learn more about functions. Because functions are so important in Python, most chapters will explore some interesting use of functions and/or introduce some interesting new language feature related to functions.

This chapter covers reading and understanding function headers, and how to call functions (including how to match arguments to parameters)

In this chapter we will learn how to read and understand function headers, and how to call functions (and methods) according to their headers. In the process, we will discuss the distinction between parameters and arguments, and Python's rules for matching them. We will illustrate these general topics with actual Python functions imported from a variety of modules as well as functions defined in the `int` and `str` classes. In later chapters we will learn how operators are defined in terms of functions, how to define functions in modules, and how to define classes that define functions.

Functions embody one important form of abstraction: the name of a function abstracts/hides the complexities of the Python statements that define the function.[1] To call/perform a function, we need to know only its name and the values it needs/uses to compute its result. In this chapter we will focus on how to understand functions defined in modules and classes that we import, and how to call these functions; in later chapters, after we learn more Python statements, we will learn how to define functions using these statements in their bodies.

A function name abstracts/hides the details of its implementation: we can call/perform a function by knowing only its name and the values it operates on

4.2 Function Headers

We characterize a function by its input (the value(s) it needs to compute the function) and it output (the value that is the result of the computation). In fact we will discuss three different ways to characterize functions, each emphasizing a different aspect of the function: headers, semantics, and bodies. As a concrete example, we will characterize the `distance` function that computes the distance between two points in the plane, using the x and y coordinates of each point.

We characterize functions by their headers and semantics: specifying what information goes into it what information comes out of it

In the process, we will also start to become familiar with three interrelated technical terms: **argument**, **parameter**, and **returned result**. An argument is a value (any object) supplied to a function as an input; arguments are also supplied to operators, although in that context they are often called operands. A parameter is a name that the function defines (in the namespace of a function object) to store a reference to an argument, so the argument can be referred to while computing the function. We explore **matching** or **passing** arguments to parameters: each parameter is added to the namespace of the function and is bound to its matching argument. Finally, a function **returns** a reference to a value (any object) that is the **result** of its computation.

When a function is called Python binds the parameters (names) in its header to the arguments (values) in the call, and computes the result (value) it returns based on these arguments

- The **Header** (form) of a function specifies the name of the function and the name of each of its parameters. Each parameter name can be followed by an optional annotation of its type and an optional default value; the header can also optionally specify `->` followed by the type of the result returned by the function. This information also communicates the number, order, and type of parameters. The header for `distance` is:
  `distance(x : float, y : float, x_ref : float = 0.0, y_ref : float = 0.0) -> float`

- The **Semantics** (meaning) of a function specifies the relationship between the arguments that are the inputs to a function call and the result returned by the function as its output. The semantics that characterize the `distance` function, in English, are: `distance` computes the euclidean distance between the point `(x,y)` and the point `(x_ref,y_ref)` specified by the arguments matching these parameters (with default values of `0.0` for the last two parameters if their matching arguments are omitted).

- The **Body** of a function specifies the Python statement(s) that implement the function's semantics. The body of the `distance` function is a single statement: `return math.sqrt( (x-x_ref)**2 + (y-y_ref)**2 )`

[1] This is similar to how the left–hand side name of an EBNF rule defines a complex control form on its right–hand side. In EBNF we use the name of the rule as a shorthand for its control form in the right–hand side of other rules. In Python we use the name of the function to call the function and compute its result by executing all the statements in its definition.

A function header and a function call both specify comma-separated information inside parentheses

To **Call** a function, we specify the name of the function and the arguments on which the function computes. We will soon explore in detail how Python matches the argument values in a function call to the parameter names in its header, discussing matching by position, by name, and including the use of the default arguments for parameters specified in the header. For example, one call of the `distance` function is `distance(3.0, 5.0, 1.0, 1.0)` which computes the distance from the point `(3.0,5.0)` to the point `(1.0,1.0)`. Notice that the function header and the function call both enclose comma–separated information inside a pair of open–/close–parentheses.

Although we will focus on reading/understanding function headers and calling their functions correctly, this paragraph shows one Python definition of the `distance` function

In this chapter we will focus on how to read and understanding function headers, and how to call the functions they describe. Of course, we also need to know the semantics of a function to understand when to use it. We typically write the semantics in English, mathematics, pictures, or whatever provides a short and unambiguous explanation of the relationship between the arguments on which a function call operates and the result that a function call returns. We defer our study of function bodies, until we learn more about the Python statements used in their definitions. But as a preview, here is how we might define the `distance` function in Python, including a triple–quoted comment that documents this function and shows some sample arguments and the results that `distance` computes, in a form similar to that which we saw for the interpreter: e.g., the triple–chevron prompt `>>>` with the returned result on the next line.

```
1  def distance(x : float, y : float, x_ref : float = 0.0, y_ref :float = 0.0) -> float:
2    """
3    Computes the euclidean distance between the point (x,y) and the point
4    (x_ref,y_ref) specified by the arguments matching these parameters (with
5    default values of 0.0 for the last two parameters, if their matching
6    arguments are omitted).
7    >>> distance(0.0, 0.0)            #use default values for x_ref and y_ref
8    0.0
9    >>> distance(0.0, 0.0, 1.0, 1.0)  #supply arguments for each parameter
10   1.4142135623730951
11   >>> distance('a','b','c','d')     #violation of the header's type annotations
12   Traceback (most recent call last):
13     ...
14   TypeError: unsupported operand type(s) for -: 'str' and 'str'
15   """
16
17   return math.sqrt( (x-x_ref)**2 + (y-y_ref)**2 )
```

We start our study of function headers by examining the EBNF rules for writing them

The rest of this section presents the EBNF for function headers, along with a few examples. Then, the next section explains how we use our knowledge of a function's header to write correct calls to the function: specifically, how the argument information in a function's call is matched to the parameter information in a function's header.

Headers document functions with their names; their parameter names, types, and default arguments; and the function's return type

Headers document functions by specifying information about their names, parameters, and return value, using the following EBNF[2] When we learn to define functions, we will see a very similar EBNF rule as a major part of the *function_definition* EBNF rule.

[2]Omitted from this EBNF description: combined dictionary parameters and annotations that are arbitrary objects, not just objects representing types.

EBNF Description: *function_header*

type_name ⇐ `int`|`float`|`imaginary`|`bool`|`str`|`bytes`|`NoneType`|`object`| any other imported/defined type/class
annotation ⇐ `:`*type_name*
default_argument ⇐ `=`*expression*
parameter ⇐ *name*[*annotation*][*default_argument*] | `*`[*name*[*annotation*]]
function_header ⇐ *qualified_name*`(`[*parameter*{`,`*parameter*}])`

The *type_name* rule includes all the type/class names we know and will learn; when we use the name `object` as a *function_header* it means a reference to any type of Python object. There are two special syntax constraints for *function_header* that pertain to the `*` option in the *parameter* rule: the `*` can appear by itself or optionally be followed by a *name*:

There are two syntax constraints related to parameters, not specified in the EBNF

1. We can use the second option in the *parameter* rule at most one time.
2. All the parameters that discard *default_argument* must appear before all the parameters that include *default_argument*, although both can appear in any order after a `*` parameter.

We could encode these requirements in a more complex EBNF rule, but deem it better to write a simpler EBNF rule here and supply this constraint verbally. We must pay attention to these two rules only when we start writing our own functions, because all the functions we study from the standard Python modules already satisfy these requirements.

All functions already defined in Python modules satisfy these two syntax constraints

An optional *annotation* indicates the type of the argument that must match the type specified for that parameter. An optional *default_argument* indicates the value that will match that parameter, if no argument explicitly matches it. The `*` alternative in the *parameter* rule specifies a special kind of parameter that can match multiple (zero or more) arguments. Although parameters often specify type annotations and default arguments, they may omit this information. For example, without any of these options, we could write the header of the `distance` function as just `distance(x, y, x_ref, y_ref)`

Annotations and default arguments are optional; `*` specifies a special parameter that can match multiple arguments

Below are six simple but illustrative examples of functions and their headers. These descriptions also include a brief semantic description of each function, to make each more understandable. The next section will show and discuss actual function calls for these functions, further illustrating their headers and semantics. Good names for functions and parameters can greatly aid us when we are trying to understand a function and pass the correct arguments to it.

Six examples of function headers and their semantics; good names can help us understand functions more easily

- `math.factorial(x : int) -> int`
  defines the header of the `factorial` function that is defined in the `math` module. It specifies that this `factorial` function has one `int` parameter and returns an `int` result.

  Semantically, it returns $x!$ (the product $1 \times 2 \times ... \times x$). Note that the name `x` is generic, indicating there is nothing special to communicate about it: other simple generic names for *int* parameters are `i`, `n`, etc.

- `random.random() -> float`
  defines the header of the `random` function that is defined in the `random` module. It specifies that this `random` function requires zero/no parameters (discard the option in *function_header*) and returns a `float` result.

  Semantically, it returns a random number as a result, whose value is uniformly distributed in the interval $[0, 1)$ meaning the value of the returned result is always ≥ 0 and strictly < 1.

If a function has no parameters in its header, we call it with no arguments, but the parentheses are always present in a function header and its call

- `distance(x : float, y : float, x_ref : float = 0.0, y_ref :float = 0.0) -> float` defines the header of the `distance` function. It specifies that this `distance` function requires four `float` parameters (the last two of which specify *default_argument*s) and returns a `float` result.

  Semantically, it returns the euclidean distance between the point at coordinate `(x,y)` and the point at coordinate `(x_ref,y_ref)`.

- `builtins.print(*args : object, sep : str = ' ', end : str ='\n') -> NoneType`[3] defines the header of the `print` function that is defined in the `builtins` module (and thus automatically imported into the script and all other modules). It specifies that this `print` function has one special `*` parameter that matches zero or more `object` values, followed by two more `str` parameters that specify default arguments, and returns a `NoneType` result: which means it returns `None` because that is the only value in the `NoneType` class.

  Semantically, it prints on the console all values matched/passed to `*args`, printing the `sep` value between each pair of values and printing the `end` value at the end: the default argument `'\n'` is an escape sequence that means advance to the next line. The `print` function returns no interesting value: its purposes is to affect the console by printing information there; but all functions must return some value, so `print` returns the value `None`.

- `builtins.str.find(self : str, sub: str, start : int = 0, end : int = None) -> int` defines the header of the `find` function that is defined in the `str` class that is defined in the `builtins` module. It specifies that `find` requires two `str` parameters and two `int` parameters (which specify *default_argument*s) and returns an `int` result. Although the `end` parameter is annotated by the type `int`, it also works correctly when passed the `NoneType` value `None`, its default argument. The parameter name `self` has a special meaning that we will explore when we discuss calling methods.

  Semantically, if all the characters in `sub` occurs in a sequence in `self` between indexes `start` and `end+1` (where `None` for `end` means there is no end value that limits the ending index), this function returns the lowest found index; otherwise it returns `-1`. Indexes in Python start at `0`, not `1`.

- `builtins.str.strip(self : str, remove : str = ' ') -> str` defines the header of the `strip` function that is defined in the `str` class that is defined in the `builtins` module. It specifies that `strip` requires two `str` parameters (the last of which specifies a *default_argument*) and returns a `str` result.

  Semantically, `strip` returns a string containing the characters in `self` in the same order, but with all the characters in `remove` stripped from the front and rear (not appearing in the string).

A function header supplies all the information needed to write a correct function call; Python reports incorrect calls by raising an exception

Regardless of the semantics of these functions, their headers specify all the information we must know to call them correctly. When we explore calling functions in the next section, we will learn how Python reports errors in calls to functions in which the arguments don't correctly match their headers: as you might expect, Python raises an exception in such cases.

[3] The `print` function actually has a fourth parameter specified by `file = sys.stdout` that specifies where to print its information. Its default argument, the console, is used frequently.

Section Review Exercises

1. Write headers for each of the functions described below. Specify appropriate names with annotated types based on these descriptions; don't write any default arguments.
   a. A function that counts the number of primes in a range between two integers, exclusive; for example there are `2` primes between `15` and `20`: `17` and `19` are both prime.
   b. A function that determines whether or not two points lie in the same quadrant; each point is specified by two `float` values representing its x and y coordinates.
   c. A function that determines whether or not two lines intersect; each line is specified by two `float` values: its slope and its Y intercept.
   d. A function that determines the majority value among three `bool` values: at least two must have the same truth value value (maybe all three).
   e. A function that selects those characters from one string that are specified in another (discarding the others); for example, when selecting from `'Able was I, ere I saw Elba'` using `'aeiou'` the result returned is `'AeaIeeIaEa'`, where case is unimportant for the selection process.
   f. A function that determines the number of fractions between two integers (inclusive) whose numerator and denominator are no larger than a third integer; for example, the number of fractions between 2 and 4 whose numerators and denominators are no larger than 5 is 5: 2/1, 3/1, 4/1, 4/2, 5/2, when the fraction 2/1 is considered different than 4/2.
   g. A function that interleaves characters in two strings; for example, when interleaving `'abcd'` with `'wxyz'` the result is `'awbxcydz'`.

   Answer:
   a. `count_primes(low:int,high:int)->int`
   b. `same_quadrant(x1:float,y1:float,x2:float,y2:float)->bool`
   c. `intersect(slope1:float,y_intercept1:float,slope2:float,y_intercept2:float)->bool`
   d. `majority(b1:bool,b2:bool,b3:bool))->bool`
   e. `select(text:str,selections:str)->str`
   f. `number_of_factions(low:int,high:int,max:int) -> int`
   g. `interleave(s1:str,s2:str) -> str`

2. What syntax constraint is violated by the following header: `f(x=1,y)`

   Answer: Omitting the annotations for the parameter and return types is allowed, but rule 2 (after the EBNF rules) is violated: a parameter without a default argument appears after a parameter specifying a default argument. The next section shows why this is problematic.

4.3 Function Calls: Matching Arguments to Parameters

The syntax for function calls is similar to the syntax for function headers

All function calls match/pass their arguments to the parameters defined in that function's header, compute the result of the function, and return that result. Function calls specify their arguments by using the following EBNF[4]

[4]Omitted from this EBNF description: combined dictionary parameters. Calls match headers, so for now both omit the same information from their EBNF.

EBNF Description: *call* (and extending *expression*)

expression ⇐ *literal* | *qualified_name* | *call*
argument ⇐ [*name*=]*expression*
function_call ⇐ *expression*([*argument*{,*argument*}])
method_call ⇐ *expression.name*([*argument*{,*argument*}])
call ⇐ *function_call* | *method_call*

Function calls typically start with a reference to a function object; method calls typically start with a reference to a value object, which is followed by a dot and a function name defined in the class of the value object

In this EBNF, both function calls and method calls start with an *expression*. For function calls, this *expression* is typically a qualified name that refers to a function object defined in some module: e.g, `math.factorial` assuming `import math`, or `factorial` assuming `from math import factorial`, or `fact` assuming `from math import factorial as fact`. For method calls, this *expression* is typically a qualified name that that refers to a value object and then (after the dot) refers to a function object defined in its class: e.g, `s.find`, assuming we defined `s` to refer to some `str` object and `find` is defined in the `str` class.

Following a reference to a function/method object by matching parentheses instructs Python to call the function/method object with the argument values in the parentheses

When we write a *function_call* or *method_call*, the required matching parentheses tell Python to call the function or method object specified right before them. Inside these parentheses, each *argument* is an *expression*, optionally preceded by *name*= where *name* must be the name of a parameter in the specified function's header. In this section we discuss calling only functions and how to match their arguments to parameters. In the next section we extend our knowledge to calling methods, which are special functions governed by the fundamental equation of object–oriented programming.

Classifications for arguments and parameters

Before exploring the argument–parameter matching rules themselves, we first classify arguments and parameters, according the options they include.

- Arguments
  - positional–argument : an *argument* **not** preceded by *name*= option
  - named–argument : an *argument* preceded by *name*= option
- Parameters
  - name–only parameter: a *parameter* discarding *default_argument*
  - default–argument parameter: a *parameter* including *default_argument*

When we call a function, Python defines each parameter name in the function object's namespace and binds each parameter name to its matching argument value object, using the rules M1–M5

When Python calls a function, it defines (in the namespace of the function object) every parameter name in the function's header, and binds to each (just like an *assignment_statement*) the argument value object matching that parameter name. In the rules below, we will learn how Python matches arguments to parameters according to three criteria: positions, parameter names, and default arguments for parameter names. Here are Python's rules for matching arguments to parameters.

M1. Match positional–argument values in the call sequentially to the parameters named in the header's corresponding positions (both name–only and default–argument parameters are OK). Stop when reaching any named–argument in the call or the * parameter in the header.

M2. If matching a * parameter in the header, match all remaining positional–argument values to it. When we learn about tuples in Section **??**, we will see that Python creates a tuple for these arguments: a tuple is just a list of values separated by commas inside matching parentheses.

M3. Match named–argument values in the call to their like–named parameters in the header (both name–only and default–argument parameters are OK)

M4. Match any remaining default–argument parameters in the header, unmatched by rules M1 and M3, with their specified default arguments.

M5. Exceptions: If at any time (a) an argument cannot match a parameter (e.g., a positional–argument follows a named–argument) or (b) a parameter is matched multiple times by arguments; or if at the end of the process (c) any parameter has not been matched, raise an exception: `SyntaxError` for (a) and `TypeError` for (b) and (c). These exceptions report that the function call does not correctly match its header[5]

Passing arguments to parameters is similar to executing a sequence of *assignment_statement*s that each bind a parameter name to its matching argument value

When this argument–parameter matching process if finished, Python defines, in the function's namespace, a name for every parameter and binds each to the argument it matches using the above rules. If an argument was itself a name, then the value object it is bound to will be shared by the parameter name while the function executes, just like sharing in the *assignment_statement* rule. In fact, passing parameters is similar to performing a series of *assignment_statement*s between parameter names and argument values.

If Python raises no exception during parameter binding, then each parameter in the function's header is bound to exactly one argument

If a function call raises no exception, these rules ensure that each parameter in the function header matches the value of exactly one argument in the function call. After Python binds each parameter name to its argument, it executes the body of the function, which computes and returns the result of calling the function; of course, the code inside function bodies can also raise exceptions.

Figure 4.1 illustrates the rules for matching arguments to their parameters in function calls

Figure 4.1 illustrates successful and unsuccessful function calls, using some of the function headers described in the previous section. The second column indicates which arguments match which parameters (and the rules used for each matching); unsuccessful matches indicate which part of rule M5 is violated.

Figure 4.1: Sample Function Calls: Matching Arguments to Parameters

Call	Parameter/Argument bindings(matching rule) and/or Exception
`math.factorial(5)`	`x=5`(M1)
`math.factorial(x=5)`	`x=5`(M3)
`distance(3.0,5.0)`	`x=3.0,y=5.0`(M1) `x_ref=0.0,y_ref=0.0`(M4)
`distance(x_ref=3.0,y_ref=5.0,x=0.0,y=0.0)`	`x_ref=3.0,y_ref=5.0,x=0.0,y=0.0`(M3)
`print('a','b','c',sep=':')`	`*args=('a','b','c')`(M2) `sep=':'`(M3) `end='\n'`(M4)
`math.factorial()`	`TypeError`(M5c:x)
`math.factorial(1,2)`	`x=1` `TypeError`(M5a:2)
`distance(1.0,2.0,x=1.0)`	`x=1.0,y=2.0` `TypeError`(M5b:x)

We can write a function definition that prints its arguments to test argument–parameter matching in the Python interpreter

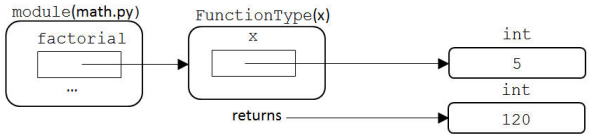
We can use the interpreter to define a function whose body just prints the values passed to its parameters, and then explore argument–parameter matching in function calls to it. For example, we can define the function `f` (without type annotations) and its body, which just prints each of its arguments: `>>> def f(a,b=2): print(a,b)`. Then calling `>>> f(1)` displays `1 2` and calling `>>> f()` displays an error message whose last line is `TypeError: f() takes at least 1 argument (0 given)`. We can enter similar function definitions in the interpreter to explore more complex argument–parameter matching in more complex function calls.

[5]Calling it a `HeaderError` exception makes more sense; but Python uses `SyntaxError` and `TypeError`. If we think of the header as specifying the type of the function —the names and types of all its arguments— then `TypeError` makes some sense.

We illustrate pictures of calling function objects on the top with their header and on the bottom with their return reference

Figure 4.2 illustrates how we picture calling a function object. It has a namespace for its parameters, and we annotate it on the top with the the type `FunctionType` and the function's parameter names and on the bottom with `returns` referring to the object it returns as a result.

Figure 4.2: Illustrating a Function Call: `math.factorial(5)`



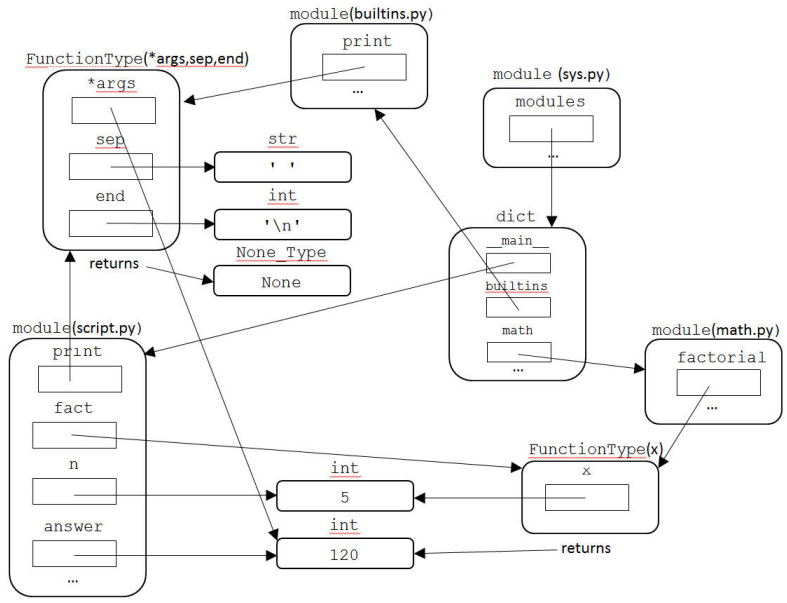
A complex picture of objects used in a small script

Figure 4.3 illustrates all the details of how we picture Python executing a script that imports a reference to the `math.factorial` and `builtins.print` function objects, computes 5! and prints the result on the console. We could also enter each of these statements into the Python interpreter.

`script` module

```
1 from math import factorial as fact     #use a fancy import form
2 n = 5
3 answer = fact(n)
4 print(answer)
```

Figure 4.3: Illustrating a Script calling Functions



The details explaining how Python executes this script

When Python executes this script it prints 120 on the console. Let's explore all the details of how this happens.

- When the script runs, the `sys` module binds its `modules` name to an empty `dict` and then adds `__main__` to that namespace, bound to the module object for `script`.

- The `builtins` module is automatically imported into the `script` module: `builtins` is added to the namespace of `dict`, bound to that module object; also, all its names are imported into `script` although we show only the name `print` in the module object for `builtins` and `script`: that is the only one that is used in `script`.
- Python then starts executing statements in `script`.
- Statement 1 imports `factorial` from the `math` module as the name `fact`: `math` is added to the namespace of `dict`, bound to the module object of `math`, whose namespace contains the name `factorial` bound to that function's object (among many other definitions not used/shown). The name `fact` is added to the namespace of the `script` module object, and bound to/shares this same function object.
- Statement 2 adds `n` to the namespace of `script`, bound to the `int` value object `5`.
- Statement 3 executes an *assignment_statement* calling the function object that `fact` refers to: Python adds the parameter `x` to this function object's namespace, bound to/sharing the same value object that its matching argument `n` refers to; this function then computes and returns as its result a reference to the `120` value object, which is bound to `answer` after `answer` is added to the namespace of the `script` module object.
- Statement 4 calls the function object that `print` refers to: Python adds its parameters `*args` (still a bit magical), `sep`, and `end` to the namespace of `print`, bound to their matching argument value objects; this function prints `120`, on the console and then terminates that line (so any future printing in the console would start on the next line) and returns a reference to the value object `None` (which `print` always returns).

We can perform the computation from the previous script more simply in a shorter script using function composition

We could compute/print the same result with the following smaller script. When executed, the result returned from calling `math.factorial(5)` —the reference to the value object `120`— is bound to the `*args` parameter in the `print` function. This is an example of function "composition": a function call whose argument is computed from the result returned by another function call; Python allows these nested/composed function calls because *expression* now has `call` as one of its options, and every argument is an expression that can be passed to a parameter.

`script` module

```
1 import math                   #use a simple import
2 print(math.factorial(5))      #print result returned by calling function on 5
```

Python does not automatically check that arguments match the annotated types of their matching parameters

As a final note on calling functions, when the parameters of a function header are annotated with types, Python does not automatically check that the matching arguments are of these types: it does not immediately report a problem with the function call if they do not match. Instead, it binds the parameters to the arguments and then executes the statements in the body of the function.

If an argument doesn't match the annotated type of its parameter, typically the function's body will malfunction and raise an exception

Typically if any argument's type does not match its parameter's type annotation, the statements in the body of the function will malfunction and cause Python to report an error by raising an exception —but the exception is raised

only after the function starts executing: so the error message may be confusing, because it is not about the function call itself, but about the code inside the function; and this code is supposed to be hidden behind the function abstraction.

When we write a function, we can put an `assert` statement in its body to raise an exception if any argument doesn't match the annotated type of its parameter

When we learn how to write functions in Chapter **??** we will see how they use `assert` statements, which often appear as the first statements in a function body. These assertions can check the type of each parameter (and other required properties) and raise an `AssertionError` exception, if any type is incorrect: e.g. factorial can check that its parameter is a non-negative integer (using the `in`, `and`, and `>=0` operators we discuss in the next chapter)

```
assert type(x) is int and x >= 0, 'argument matching x is incorrect'
```

By using a decorator, we can explicitly instruct Python to check annotations when function is called, raising an exception if any argument is incorrect, before executing the body of the function

Furthermore, when we learn how to use decorators in Chapter **??**, we will discuss how to generalize annotations beyond just types (e.g., we can specify, as we did in the `assert` above, that a parameter should be an `int` whose value is ≥ 0) and how to tell Python to perform the annotation checks on a function whenever it is called; these can raise an exception immediately indicating a problem —before Python executes the body of the function— if the argument's value does not satisfy the parameter's annotation.

The value `None` can often be passed to some parameter whose type annotation is not `NoneType`; in such cases, often `None` is the default argument for that parameter

Finally, sometimes we can successfully pass the value `None` to a parameter whose type is not `NoneType`. For example, in the header for `buitins.str.find` in Section 4.2 the `end` parameter specifies an `int` upper–bound on the indices to check; but if there is no upper bound (all indices are checkable), we can pass the value `None` to this parameter and the function will execute properly: in fact, `None` is its default argument.

4.3.1 Assigning and Printing Function Objects

In Python, we can bind a name to any type of object: we have studied module objects, value objects, and function objects

We have learned that names refer to objects: we have studied how names are defined and bound to objects with import statements, assignment statements, and in function calls when matching/passing arguments to parameters. We have seen three types of objects: module objects, value objects, and function objects (and will soon study the fourth kind: class objects). Python's assignment statement and parameter binding mechanism allows us to bind any name to any object. So, for example, we can write the assignment statement `fact = math.factorial` which defines the name `fact` and binds it to the same object `math.factorial` refers to: both names now share this function object. This is similar to writing `from math import factorial as fact`.

Python prints a value object by displaying a string representing the value stored in the object

When we call the `print` function to print a value object, Python displays on the console a string representing the value stored in the object. For the types of value objects we have studied, most display using the same EBNF that describes their literal values: e.g., `print(1)` displays `1`. Likewise, in the Python interpreter if we enter `>>> 1` Python's read–execute–print loop automatically displays `1`. Most other types, including `float` and `bool`, display the same way.

Python prints a string value by displaying its characters without enclosing quotes

But when we call the `print` function on `str` value objects, they display their text without their outer quotes: e.g., `print('acggta')` displays `acggta`; although in the interpreter, if we enter `>>> 'acggta'` Python displays `'acggta'` (showing the quotes); in fact if we enter `"acggta"` Python also displays `'acggta'`,

always displaying single–quotes.[6] If we call the `print` function in the interpreter, it displays its argument values using the standard `print` semantics: e.g., `>>> print('acggta')` displays `acggta`.

If the result of the interpreter's >>> prompt is None, the interpreter does not print this value, but just reprompts immediately with >>>

In addition, recall that the result returned by the `print` function is `None`, and the Python interpreter does not display that value if it is the result returned by what the user entered to the prompt. So, entering `>>> None` prints nothing and reprompts immediately with `>>>`. But by entering `>>> print(None)`, the `print` function will be called and display its `None` argument, but the interpreter will not display the `None` value returned as a result by `print`; instead it reprompts with `>>>`.

When Python prints a function object, it displays the function object as a special string

When we call the `print` function on a function object, it displays the function object in a special way: e.g., `print(print)` calls the `print` function to display the function object bound to the name `print`, which displays `<built-in function print>`; likewise, calling `print(math.factorial)` displays the function object bound to the name `math.factorial` (assuming `import math`), which displays `<built-in function factorial>`. Although this function is imported from the `math` module, not the `builtins` module, it is still part of the standard Python library and displays with the words `built-in`. The interpreter displays function objects identically: e.g., `>>> print` (which does not call the `print` function object —it is not followed by parentheses) displays the function object `print` refers to: `<built-in function print>`.

A name bound to a function object refers to that object; but a reference to a function object followed by parentheses (enclosing its arguments) represents a call to that function object

So, when we see/use the name bound to a function object, we must understand whether or not we are calling the function object it refers to. We call a function object if we follow the reference to it by parentheses enclosing its argument values: calling some functions, like `random.random` use no arguments but still must have parentheses. Figure 4.4 shows an annotated interaction with the Python interpreter, illustrating how the names that refer to function objects can be used to print and call the function object; examine it closely.

Section Review Exercises

1. Given the function headers `f(a,b,c=10,d=None)`, `g(a=10,b=20,c=30)`, and `h(a,*b,c=10)...` fill in the following table using the same information displayed in Figure 4.1.

Call	Parameter/Argument bindings(matching rule) and/or Exception
`f(1,2,3,4)`	`a=1`, `b=2`, `c=3`, `d=4`(M1)
`f(1,2,3)`	`a=1`, `b=2`, `c=3`(M1) `d=None`(M4)
`f(1,2)`	`a=1`, `b=2`(M1) `c=10`, `d=None`(M4)
`f(1)`	`a=1`(M1) `c=10`, `d=None`(M4), `TypeError`(M5c:`b`)
`f(1,2,b=3)`	`a=1`, `b=2`(M1) `b=3`(M3), `TypeError`(M5b:`b`)
`f(d=1,b=2)`	`d=1`, `b=2`(M3) `c=10`(M4), `TypeError`(M5c:`a`)
`f(b=1,a=2)`	`b=1`, `a=2`(M3) `c=10`, `d=None`(M4)
`f(a=1,d=2,b=3)`	`a=1`, `d=2`, `b=3`(M3), `c=10`(M4)
`f(c=1,2,3)`	`c=1`(M3), `SyntaxError`(M5a:2)
`g()`	`a=10`, `b=20`, `c=30`(M4)
`g(b=1)`	`b=1`(M3), `a=10`, `c=30`(M4)
`g(a=1,2,c=3)`	`a=1`(M3), `SyntaxError`(M5a:2)

[6] In Section ?? we will discuss the difference between the `__str__` and `__repr__` functions on value objects. The `print` function calls `__str__` to display its arguments; the read-execute-print loop calls `__repr__`. These functions can return slightly different information as strings.

Figure 4.4: Examples of Printing and Calling Function Objects

Entered Statement	Annotation
`>>> from math import factorial`	Import `factorial` function: returned result is non–printing `None`
`>>> factorial` `<built-in function factorial>`	Name `factorial` refers to a function object Print the function object bound to the name `factorial`
`>>> factorial(5)` `120`	Call the function object bound to the name `factorial` using the argument `5`; print the returned result
`>>> x = factorial`	Bind `x` to the function object `factorial` refers to (share it): returned result is non–printing `None`
`>>> x` `<built-in function factorial>`	Name `x` refers to a function object Print the function object bound to the name `x`
`>>> x(5)` `120`	Call the function object bound to the name `x` using the argument `5`; print the returned result
`x = 1`	Bind `x` to the value object `1`: returned result is non–printing `None`

`>>> x(5)`
`Traceback (most recent call last):`
`  File "<stdin>", line 1, in <module>`
`TypeError:  'int' object is not callable`
	Call the function object bound to the name `x`
	But `x` refers to an `int` value object, not a function object
	So Python raises the `TypeError` exception, indicating
	that it cannot call an `int` value object

`>>> print(x)` `1`	Call print on the value object bound to the name `x` Print the argument: returned result is non–printing `None`
`>>> print(print)` `<built-in function print>`	Print the function object bound to the name `print` Print the argument: returned result is non–printing `None`

Call	Parameter/Argument bindings(matching rule) and/or Exception
`h(1,2,3,4,5)`	`a=1`(M1), `b=(2,3,4,5)`(M2), `c=10`(M4)
`h(1,2,3,4,c=5)`	`a=1`(M1), `b=(2,3,4)`(M2), `c=5`(M4)
`h(a=1,2,3,4,c=5)`	`a=1`(M3), `SyntaxError`(M5a:1)
`h(1,2,3,4,c=5,a=1)`	`a=1`(M1), `b=(2,3,4)`(M2), `c=5`(M3), `TypeError`(M5b:a)

2. What does the Python interpreter display when each of the following function calls is entered: (a) `>>> print(print(print))` (b) `>>> factorial(factorial)`. Assume `from math import factorial`.

   ANSWER:

```
>>> print(print(print))
<built-in function print>          displayed by call to inner–print
None                               displayed by call to outer–print
>>> factorial(factorial)           argument not an int object
Traceback (most recent call last): error occurs inside the function
  File "<stdin>", line 1, in <module>
TypeError: an integer is required
>>>
```

3. Describe the different between the function calls `print(abc)` vs. `print('abc')`.

   ANSWER: `print(abc)` calls the `print` function object on the object that the name `abc` refers to and returns `None`; what it displays on the console depends on what type of object the name `abc` refers to. `print('abc')` calls the `print` function object on the string value object storing `'abc'`, which always displays `'abc'` on the console and returns `None`.

4.4 Method Calls and the Fundamental Equation of Object–Oriented Programming

Methods are functions defined in classes

Recall that modules can define functions and classes; and classes can themselves define functions. We will use the technical term **method** when we explore calling functions defined in classes. We call functions and methods slightly differently in Python, but we will learn how to translate any method call into an equivalent function call simply, by using the fundamental equation of object–oriented programming.

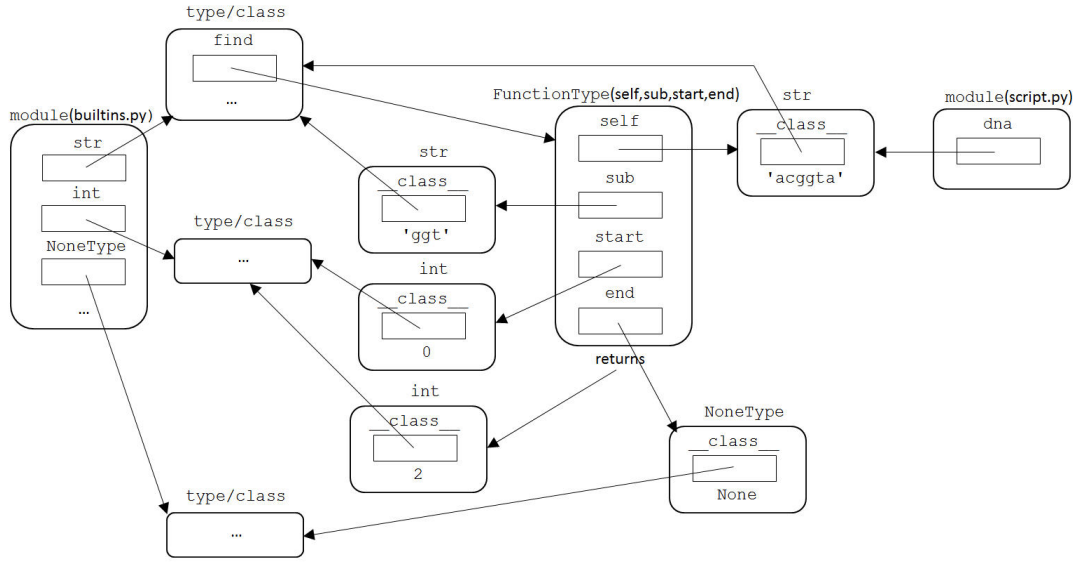
Methods are called by writing a reference to a value object; a dot; the name of a function defined in the object's class; matching parentheses enclosing arguments

Review the header of the `builtins.str.find` function presented in Section 4.5; it describes the `find` function, which is defined in the `str` class, which is defined in the `builtins` module, which is automatically imported into every module (`from builtins import *`). Assume we define `dna = 'acggta'`. To call this function as a method (e.g., `dna.find('ggt')`) we start with a reference to an object, which is followed by the method name (the two are separated by a dot) defined in that object's class, which is followed by arguments inside matching parentheses that always indicate a call. Python binds the object specified before the dot to the first parameter in the function header, and then matches all the arguments in parentheses to the remaining parameters.

Illustrating the method call `dna.find('ggt')`

Figure 4.5 illustrates the method call `dna.find('ggt')`. In this picture, all the value objects of type/class `str`, `int`, and `NoneType` are enhanced to show the special `__class__` reference in their namespaces; each refers to its class object, which the names `str`, `int`, and `NoneType` in the namespace of the `builtins` module object also refer to/share. Note that the namespace of the `str` class object shows the name `find`, which refers to one function object it defines.

Figure 4.5: Illustrating a Method Call: `dna.find('ggt')`



In a method call, Python binds the value object appearing before the method name to the first parameter in the function call (commonly named `self`)

In the method call `dna.find('ggt')`, Python first locates the `find` function using the `__class__` reference of the value object `dna` refers to (it is a `str`). In that method Python binds the first parameter (named `self`) specially to the

same object `dna` refers to. Then it binds the second parameter (named `sub`) to an object for the `str` literal `'ggt'`, which is the first actual argument in the method call. Then it binds `start` to its default argument —an object for the `int` literal `1`— and `end` to its default argument —an object for the `NoneType` literal `None`. Finally, it calls the `find` function, which returns the index 2; recall that string indexes start at `0`, so `dna` stores `a` at index `0`, `c` at index `1`, `g` at index 2: the starting index for the substring `'ggt'`.

The method call `dna.find('ggt')` is equivalent to the function call `str.find(dna,'ggt')`

The . method call `dna.find('ggt')` can be equivalently written as the function call `str.find(dna,'ggt')`. Given this particular equivalence, we are one step away from stating the fundamental equation for object–oriented programming: how to translate any method call into its equivalent function call.

The `type` function returns a reference to the type of an object, its class: the object its `__class__` name refers to

But first we need to learn one more thing about Python: its `builtins` module defines a `type` function whose parameter is a reference to an object and whose returned result is a reference to the type/class object of its parameter. It returns the reference in the storage cell for the `__class__` name in the namespace for any value object parameter, which is why Figure 4.5 shows the `__class__` name for each value object. In fact, calling the function `type(dna)` returns the same reference that `dna.__class__` returns. Note that the returned class object defines `find` in its namespace, because the `str` class defines `find`, similarly to how modules define functions.

Python translates method calls into equivalent function calls by using the `type` function

Now we can state the fundamental equation of object–oriented programming, which explains how Python translates a method call into a function call. Given any value object `v`, a method `m` (which must be defined in `type(v)`), and any number of arguments `a`, `b`, `c`, ... (which must be correct according to the header of `m`), the method call `v.m(a,b,c...)` is equivalent to the function call `type(v).m(v,a,b,c,...)` and Python translates the former into the latter automatically. Stated as an equation

Fundamental Equation of Object–Oriented Programming[7]

$$\texttt{v.m}(\texttt{a}, \texttt{b}, \texttt{c}, ...) = \texttt{type}(\texttt{v}).\texttt{m}(\texttt{v}, \texttt{a}, \texttt{b}, \texttt{c}, ...)$$

Python uses the type of the value object to find the function in the class to call, and passes that value object to the first parameter in the function

Here, Python calls the function `m` defined in the class `type(v)`; it automatically matches the special argument `v` to the first parameter, and matches all the arguments in parentheses to the remaining parameters. Since `type(dna)` refers to the same type/class object as `str`, `dna.find('ggt')` is translated by this equation into `str.find(dna,'ggt')`. The name of the first parameter in most methods is `self`, indicating it should be a reference to the value object (itself) used to call the method.

We can call method on objects created for literals

We can also use value objects for literals to call methods. For example, we can call `'acggta'.find('ggt')` in Python. Again, because `type('acggta')` is `str`, Python translates this method call into the equivalent function call `str.find('acggta','ggt')`, using this literal as the first argument in the function call.

Should we use function calls or method calls?

Given the ability to call functions and methods equivalently, which form should we use? Sometimes there is no choice: functions declared in modules must be called by their function name. But functions declared in classes can

[7] When we learn about declaring classes in inheritance hierarchies, we will discover that the rules governing this equation generalize how to find the function to call: first examining the type/class of the value object (as shown here), and if necessary its superclasses.

be called as methods or functions. Calling functions highlights the name of the function; calling methods highlights the main object being operated on by the function. Object–oriented programming puts the primary focus on the main object; it treats the method being called on that object to be of secondary importance, and treats all the other arguments (if they exist) used to control the method call to be of tertiary importance. So, generally we will call functions declared in classes as methods, prefacing each method call by the the main object on which the method operates. When we discuss inheritance, we will discover that the rules for finding the equivalent function become more complex, adding to the powerfulness of calling methods.

4.4.1 Printing Method Objects and Class Objects

Python displays method objects like function objects, but with the word `method`

We have already learned how Python displays value objects and function objects. Python displays method objects using the same left and right angle–brackets used to display function objects, but specifying the word `method` instead of `function` and displaying the type/class of the primary object on which the method can be called: `>>> str.find` displays as `<method 'find' of 'str' objects>`.

Python displays type/class objects with the word `class`

Python displays type/class objects in a form similar to function objects. For example `>>> int` returns a reference to the `int` class which displays as `<class 'int'>`, using the same left and right angle–brackets used to print function objects, specifying the word `class` and a string naming the class (`'int'`). Likewise, `>>> str` returns a reference to the `str` class which displays as `<class 'str'>`. We can refer to the names `int` and `str` because of the automatic import: `from builtins import *`. The `type` function produces similar results: `>>> type(1)` displays as `<class 'int'>` and `>>> type('acggta')` displays as `<class 'str'>`.

Two methods (`bit_length` and `count`) used in the upcoming problem set

Python also defines the following two methods, used in the problems below. First, `int.bit_length(self : int) -> int` returns the number of binary digits (bits) needed to represent the magnitude of the integer specified by its parameter: `(19).bit_length()` returns the result `5`, because $19_{10} = 10011_2$, and the binary number has 5 bits. Trying to call `'a'.bit_length()` raises `AttributeError` because the `str` class defines no `bit_length` method attribute. Second, `str.count(self : str, check : str) -> int` returns the number of times `check` occur in `self`: `'acggta'.count('a')` returns the result `2` because there are two occurrences of `'a'` in `'acggta'`.

Section Review Exercises

1. Show Python's tokenization of `19.bit_length()`; what problem occurs if we enter this information into the interpreter?

   Answer: Python tokenizes this information as $\boxed{\texttt{19.}}^{f}$ $\boxed{\texttt{bit\_length}}^{i}$ $\boxed{\texttt{(}}^{d}$ $\boxed{\texttt{)}}^{d}$: a float literal followed by an identifier. Here the dot is treated as part of a float literal, not as a delimiter between a reference to a value object and a method name, so the Python interpreter raises an exception. To fix this problem, we must use parentheses around the integer literal.

2. Translate the following method calls into equivalent function calls. Assume we have declared `s = 'bookkeeping'` and `d = 'o'`. a. `(19).bit_length()`;

b. `'acggta'.count('a')` c. `s.count(d)` d. `'acggta'.strip('ac')` –and what result does this method call return?

ANSWER: a. `int.bit_length(19)`; b. `str.count('acggta','a')` c. `str.count(s,d)` d. `str.strip('acgtta','ac')` returns `'ggt'`

4.5 Names Defined In Modules

4.5.1 Names Defined in the `builtins` Module

We classify the names imported from the `builtins` module into three tables: types/classes, functions, and exceptions

Tables 4.1–4.3 present the names defined in the `builtins` module, organized by class names, function names, and exception names (exceptions are actually special class names). Table 4.1 presents the names of types/classes defined in the `builtins` module. These include the types/classes we already know from out discussion of literals, as well as many other classes, some of which we will examine later in this book. Note that `type` is actually a class, not a function, but we can often call a class just like a function, and for `type` we have learned its semantics.

Table 4.1: Type/Class Names Defined in the `builtins` Module

`bool`	`enumerate`	`map`	`set`	`type`
`bytearray`	`filter`	`memoryview`	`slice`	`zip`
`bytes`	`float`	`object`	`staticmethod`	
`classmethod`	`frozenset`	`property`	`str`	
`complex`	`int`	`range`	`super`	
`dict`	`list`	`reversed`	`tuple`	

The `print` function displays output information on the console; the `input` function prompts the user and accepts input from the console

Table 4.2 presents the names of functions defined in the `builtins` module. We have already discussed the `print` function, which displays output on the console. In Section 4.5.3 we will discuss the `input` function, which prompts the user to type input on the keyboard With these two functions, we can write scripts that prompt for inputs and display the results of computations on these inputs.

Table 4.2: Function Names Defined in the `builtins` Module

`abs`	`dir`	`help`	`max`	`round`
`all`	`divmod`	`hex`	`min`	`setattr`
`any`	`eval`	`id`	`next`	`sorted`
`ascii`	`exec`	`input`	`oct`	`sum`
`bin`	`format`	`isinstance`	`open`	`vars`
`callable`	`getattr`	`issubclass`	`ord`	
`chr`	`globals`	`iter`	`pow`	
`compile`	`hasattr`	`len`	`print`	
`delattr`	`hash`	`locals`	`repr`	

The `help` function displays docstring information about objects –especially function objects– on the console

The `help` function is of special interest for programmers exploring Python: its prototype is `help(name : object) -> str`; semantically it returns the special "docstring" describing whatever object is supplied as its argument, which is typically a function, module, or class object (for a value object it returns the

docstring of its class, which can be lengthy). See the docstring for the `distance` function on page 55 for an example. We can call `help` in the interpreter to print useful information about any defined name. Here is an example.

```
>>> help(print)
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file: a file-like object (stream); defaults to the current sys.stdout.
    sep:  string inserted between values, default a space.
    end:  string appended after the last value, default a newline.
```

Get into the habit of using `help` to explore the headers and semantics of functions

Here, the `help` function returns a docstring, which the interpreter prints. Generally, docstrings are meant to be read by knowledgeable programmers, but it is good to get into the habit of using `help` in the interpreter to help us become more knowledgeable Python programmers. More information is available on Python's documentation home page: `www.python.org/doc/`.

Python's `builtins` module defines many exception names

Table 4.3 presents the names of exceptions defined in the `builtins` module. Notice that exceptions names all use a special convention, where the first letters in multi–word names are capitalized, called "camel notation". Every exception name is a special kind of class name, so they could appear in Table 4.1. But because there are so many exceptions, we list them here in a separate table.

Table 4.3: Exception Names Defined in the `builtins` Module

ArithmeticError	ImportError	ReferenceError	UnicodeEncodeError
AssertionError	ImportWarning	ResourceWarning	UnicodeError
AttributeError	IndentationError	RuntimeError	UnicodeTranslateError
BufferError	IndexError	RuntimeWarning	UnicodeWarning
BytesWarning	KeyError	StopIteration	UserWarning
DeprecationWarning	KeyboardInterrupt	SyntaxError	ValueError
EOFError	LookupError	SyntaxWarning	Warning
EnvironmentError	MemoryError	SystemError	WindowsError
Exception	NameError	SystemExit	ZeroDivisionError
FloatingPointError	NotImplementedError	TabError	
ExceptionFutureWarning	OSError	TypeError	
GeneratorExit	OverflowError	UnboundLocalError	
IOError	PendingDeprecationWarning	UnicodeDecodeError	

4.5.2 Functions Defined in the `math` Module

Python's `math` module defines many standard mathematical and scientific functions

Table 4.4 presents the names of functions defined in the `math` module, which provides many useful mathematical and scientific functions (most of which are found on calculators). Try importing the `math` module in the interpreter, and then typing `>>> help(math)` as well as `>>> help(math.sqrt)` to see Python's documentation of this entire module and one of its functions.

Table 4.4: Functions Named Defined in the `math` Module

acos	cos	floor	isnan	sinh
acosh	cosh	fmod	ldexp	sqrt
asin	degrees	frexp	lgamma	tan
asinh	erf	fsum	log	tanh
atan	erfc	gamma	log10	trunc
atan2	exp	hypot	modf	
atanh	expm1	log1p	pow	
ceil	fabs	isfinite	radians	
copysign	factorial	isinf	sin	

4.5.3 Input Functions in the `builtins` and `prompt` Modules

There are two ways to prompt users for input on the console, using functions defined in the `builtins` and `prompt` modules

In this section we will discuss how to perform simple input with the `input` function defined in the `builtins` module and begin exploring some of the simpler functions defined in the `prompt` module to do input with error detection and recovery. We will describe all these functions by their headers and semantics.

The `input` function prompts the user with a string and returns a string

The `input` function has the header `input(prompt : str = '') -> str`; note the default argument for `prompt` is an empty string: a string with no characters. The semantics of calling `input` are to display the `prompt` string on the console (without its quotes) and wait until the user finishes typing a response on the keyboard by pressing ↩ (echoing the characters on the console as the user enters them): the result it returns is a string containing all the characters in the user's response. So, if we wrote `last_name = input('Enter last name: ')` as a statement in a module, it could produce the following interaction on the console (with the user typing `Pattis` and pressing ↩).

```
Enter last name: Pattis
```

The `prompt` string is displayed without quotes and the user's response string is entered without quotes

This assignment statement would define the name `last_name` and bind it to the string `'Pattis'`; if we pressed ↩ immediately it would be bound to the empty string `''`. Note that prompt is displayed without quotes and the user does not type quotes for the string returned by `input`: the result returned in always a string. In fact, the following interaction on the console

```
Enter last name: 7
```

would define the name `last_name` and bind it to the string `'7'`: typing digits still results in this function returning a string, as specified by its header.

The `int` function creates an `int` object from a `str` object; we can compose function calls of `int` and `input` to enter an integer

So, how can we input an integer? We call the `int` class as a function on the result returned by `input`: e.g., `cents = int(input('Enter change: '))` in which the string returned by `input` becomes an argument to the `int` function which constructs an object of the `int` type/class whose value is specified by the digits in its string argument. This is another example of function composition. In fact, we could write the Python statement

```
print(math.factorial(int(input('Enter factorial argument: '))))
```

which is a quadruply–composed function: Python calls the `input` function and uses the result it returns as an argument to the `int` function, which uses the result it returns as an argument to the `math.factorial` function, which uses

the result it returns as an argument to the `print` function. Said another way, Python prints the factorial of the integer value of the string input.

The `int` function is overloaded: it has multiple headers

There are two useful headers for `int`. When a function has more than one header, we call it "overloaded"; there is no negative connotation to being an overloaded function in Python, overloaded is just a descriptive technical term.

- `int(value : str, base : int = 10) -> int`
  Returns the integer equivalent of its `value` parameter (a string) in the specified base (which is frequently omitted and defaults to `10`): e.g., `int('31')` returns the result `31`; and `int('101',2)` returns the result `5`, because $101_2 = 5_{10}$
- `int(value : float) -> int`
  Returns the truncated towards zero integer equivalent of its `value` parameter (a floating–point): e.g., `int(5.8)` returns the result `5`; `int(-5.8)` returns the result `-5`. So in both cases any information after the decimal point is removed to produce an `int` value object.

The `float` function is overloaded

Similarly, we can call the `float` class as a function, using either of the following headers (so `float` is also an overloaded function too).

- `float(value : str) -> float`
  Returns the floating–point equivalent of its `value` parameter (a string in the format of an optionally signed *float_literal*); e.g., `float('9.10938188E-31')` returns the result `9.10938188E-31`.
- `float(value : int) -> float`
  Returns the floating–point equivalent of its `value` parameter (an integer): e.g., `float(1)` returns the result `1.0`. We would write `1.0` not `float(1)`, but if `x` stores an `int` value object, calling `float(x)` returns a `float` value object whose value is equivalent to `x`.

In fact, Python also includes the headers `int(value : int) --> int` and `float(value : float) --> float`, which each return the same values as their argument: e.g., `int(1)` returns `1` as a result.

Experiment in the Python interpreter composing the `input`, `int`, and `float` functions

Experiment in the Python interpreter with the `input`, `int`, and `float` functions until you understand them well. Notice that if we call `int('ab')` or `int('1.2')` Python raises a `ValueError` exception, because neither string argument has an equivalent integer value, so Python must report its inability to call this function correctly with these arguments; but the composed function call `int(float('1.2'))` returns `1`, because `float('1.2')` returns `1.2` and `int(1.2)` returns `1`.

The `prompt` module defines functions that prompt the user on the console, detecting input errors and reprompting

The `prompt` module defines both general–purpose and specific functions that prompt a user for input. The functions it defines have an advantage over composing a type-conversion function with the `input` function, when the user enters input in the wrong form: they display an error messsage and reprompt the user, instead of raising an exception and terminating the script.

The headers of all the functions defined in `prompt` are similar

Here are the headers of the simplest and most commonly used functions defined in `prompt`: they are all similar. Note that the type/class `FunctionType` means that the `is_legal` parameters are bound to some function object: if not `None`, the function object specifies one parameter and returns a boolean value that determines whether or not the input is appropriate (see the semantics below, for more information).

```
prompt.for_int   (prompt_text   : str,
                  default       : int          = None,
                  is_legal      : FunctionType = (lambda x : True),
                  error_message : str          = 'not a legal value') -> int

prompt.for_float (prompt_text   : str,
                  default       : float        = None,
                  is_legal      : FunctionType = (lambda x : True),
                  error_message : str          = 'not a legal value') -> float

prompt.for_bool  (prompt_text   : str,
                  default       : bool         = None,
                  error_message : str
                    = 'Please enter a bool value: True or False') -> bool

prompt.for_string(prompt_text   : str,
                  default       : int          = None,
                  is_legal      : FunctionType = (lambda x : True),
                  error_message : str          = '') -> str

prompt.for_int_between(prompt_text   : str,
                       low           : int,
                       high          : int,
                       default       : int = None,
                       error_message : str = '') -> int
```

These functions prompt the user to enter a value, or just press ↩ to use the value of `default`

Semantically, all prompt the user by displaying the `prompt_text` string followed by `': '`; if `default` is not `None`, its value appears in square brackets between the `prompt` string and `': '` suffix. The user can then type any answer and press ↩ (Enter) (just as with the `input` function). If the user immediately presses ↩ it is as if he/she typed the argument matching `default` (whose value is shown in the square brackets).

These functions check that the user enters the right type/class of value and that `is_legal` (if it is not `None`) returns `True` for the entered value

When the user enters a string, these functions check whether it can be converted into a value of the desired type. If it can, and if the `is_legal` parameter is not bound to `None`, Python also calls the function that `is_legal` is bound to with the converted user–input value as an argument, to determine whether or not it is appropriate: the `is_legal` function returns `True` as a result. If the type/class of the input string and the optional verification are correct, the type–converted value is returned; otherwise Python prints the `error_message` string and reprompts the user (doing so any number of times, until the user enters correct input).

The difference between the `input` and `prompt.for_int` functions

Executing `cents = int(input('Enter change: '))` is similar to executing `cents = prompt.for_int('Enter change')` except if the user enters incorrect input; when calling `int` with incorrect input, Python would raise a `ValueError` exception. Here is an example of the console interaction for one call of `prompt.for_int` with incorrect input.

```
Enter change: abc
  Exception: invalid literal for int() with base 10: 'abc'
  Possible error: cannot convert str 'abc' to specified type of value
Enter change: 87
```

The user enters an incorrect type/class of value, is told about an entry error, is reprompted, and enters a correct value: 87 which is bound to cents. The user would be reprompted over–and–over (if necessary) until he/she enters a string that can be converted to an integer.

The prompt.for_int function reprompts when the user enters a value of the incorrect type

Now, suppose that we want to define a name p bound to to an integer that is prime. We are assuming that the predicate module (see below) is imported, and defines the function is_positive(i : int) -> bool: this function returns a boolean telling whether or not its argument is a positive number : strictly greater than zero. We can write a script that executes the following assignment statement

The prompt.for_int function checks the entered value using its is_legal parameter if it stores a reference to a function object (not None)

```
p = prompt.for_int('Enter a positive',17,is_legal=predicate.is_positive)
```

Here are two examples, on the left and right, of a console interaction executing this statement. Note that when calling prompt.for_int its first two arguments are matched positionally to the first two parameters, then the argument is_prime is matched to the parameter is_legal, and finally the parameter error_message is matched to its default argument 'not a legal value'.

```
Enter positive[17]: -4
  Entry Error: '-4';  not a legal value:
  Please enter a legal value
Enter positive[17]: 13
```

```
Enter positive[17]:
```

In the left interaction, p is bound to the integer 13 after the user is reprompted because the entered value -4 isn't a positive number; in the right interaction, p is bound to the integer 17: the default argument, which is also a positive number.

4.5.4 Functions Defined in the predicate Module

The predicate module defines many functions, including functions with the following headers (some overloaded). As required in the description above for functions matching the is_legal parameter, each function header specifies one argument and returns a boolean value.

The predicate module defines a variety of single–parameter functions (some overloaded) that return a boolean result

```
predicate.is_even          (i : int)   -> bool
predicate.is_odd           (i : int)   -> bool
predicate.is_positive      (i : int)   -> bool
predicate.is_non_negative  (i : int)   -> bool
predicate.is_prime         (i : int)   -> bool
predicate.is_positive      (f : float) -> bool
predicate.is_non_negative  (f : float) -> bool
....
predicate.length_equal     (i : int)   -> FunctionType
```

We can use these functions to verify input, passing any as an argument to the is_legal parameter defined in the prompt module functions. If is_legal is bound to its default argument (lambda x: True) then the input –so long as it is the correct type– is accepted. Note that the is_positive and is_non-negative (returns True for zero and positive values) are overloaded for the types int and float.

We can use the functions defined in the predicate module as arguments to the functions defined in the prompt module

The predicate module also defines the length_equal function, which has an

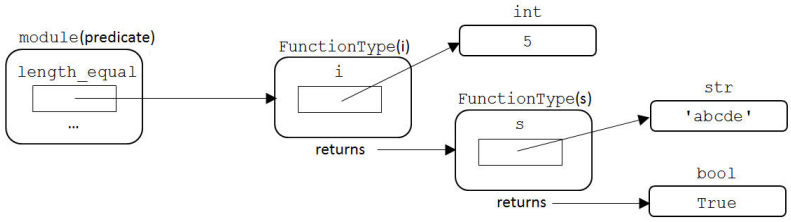
The length_equal function defined in the predicate module returns a function

interesting header: `predicate.length_equal (i : int) -> FunctionType`. This function takes an `int` parameter and returns a function as a result: semantically, the function it returns has the header `...(s: str) -> bool` where `...` indicates an unnamed function object that takes a string parameter and returns a boolean result.

So, `predicate.length_equal` is a name that refers to a function object. The call `predicate.length_equal(5)` returns a reference to another function object: one that will return `True` when its parameter is a string whose length is equal to `5`. So `predicate.length_equal(5)('abcde')` calls the returned function, which returns the result `True` for its string argument. Notice that by following `predicate.length_equal` by `(5)` we call this function; by following `predicate.length_equal(5)` by `('abcde')` we call the function it returns. Figure 4.6 illustrates this function call.

We can instruct Python to call the function that is returned by calling the `length_equal` function

Figure 4.6: The function call `predicate.length_equal(5)('abcde')`



Here is how we can use the `length_equal` function when prompting for a string: `x = prompt.for_string(is_legal=predicate.length_equal(5))`. The `verify` parameter is bound to the function returned by `predicate.length_equal(5)`, a function that returns `True` for any string of length `5`. Note that in this call to `prompt.for_string`, the `prompt`, `default`, and `error` parameters refer to their default arguments. This call could result in the following console interaction.

We can use `predicate.length_equal` to input a string that is required to contain a certain number of characters

```
Enter string: abc
Illegal string entered
Enter string: abcde
```

After this interaction, the name `x` would be bound to the string `'abcde'`.

The ability for a function call to specify another function as an argument and/or return another function as a result is a very powerful programming feature. Although this feature is simple enough to introduce and use here, we will learn more about it (including how to write functions that return functions), and see more complex uses for these features, later in this book.

Section Review Exercises

1. Entering `>>> import math` followed by `>>> help(sqrt)` in the Python interpreter fails to display information about the `sqrt` function. Write two different `import`/`help` statements that display information about the `sqrt` function. Hint: names.

   Answer: 1. `>>> import math` followed by `>>> help(math.sqrt)`
   2. `>>> from math import sqrt` followed by `>>> help(sqrt)`

2. a. Write a function call that would produce the interaction below on the

console. b. Describe the interaction if we added `default=4` in the function call and then entered ↩ immediately when prompted?

```
Enter a prime: 4
That value is not prime
Enter prime:  5
```

ANSWER: a. `prompt.for_int('Enter a prime', error='That value is not prime', is_legal=predicate.is_prime)` b. Python would display `That value is not prime` and reprompt because the `default` value specified is not prime.

CHAPTER SUMMARY

This chapter discussed how functions are called, including rules that specify how the argument values in function calls are matched/bound to the parameter names in function headers. We learned that a function header specifies the name of a function and the names, number, and order of its parameters. These parameters can be annotated with their types (as can the result returned by the function); each parameter can optionally specify a default argument that Python uses if the function call doesn't explicitly supply an argument matching that parameter. The semantics of various function headers were described in English, by specifying how their returned results (outputs) are related to their argument values (inputs); one especially important function is `print`. We saw various pictures illustrating how functions are called, and saw how function objects can be bound to names, called, and printed in Python. Next we discussed methods: functions defined inside classes. Method calls are a special kind of function call that starts with a reference to a object, followed by a function name defined for the object's class. We learned how to translate method calls into equivalent function calls, by using the `type` function, and how methods and classes are printed. All the names defined in the `builtins` class are automatically imported into every module: these names define classes, functions, and exceptions. Finally, we learned two ways to prompt the user for input on the console: using a type–conversion function composed with the `input` function, or using one of the functions defined in the `prompt` module, which are a bit more forgiving of incorrectly entered input. During our discussion of function calls we examined the composition of function calls, how functions can be passed as arguments in function calls, and even how function calls can return functions (and how these returned functions can be used).

CHAPTER EXERCISES

1. Write a script (including imports) that prompts the user for a non-negative integer and then prints that number and its factorial. An interaction might look like

   ```
   Enter x for x!: -1
   Re-enter a non-negative value
   Enter x for x!: 5
   5! is 120
   ```

2. Show what the following script displays on the console. Hint: Carefully examine the arguments matching the `sep` and `end` parameters.

```
1 print('a','b','c')
2 print('d','e','f',sep='')
3 print('g','j','i',sep=':')
4 print('j','k','l',sep='--',end=':')
5 print('m',end='\n\n')
6 print('n')
```