4. Functions

Objectives

- Create custom functions.
- Import and use Python Standard Library modules, such as `random` and `math`, to reuse code and avoid "reinventing the wheel."
- Pass data between functions.
- Generate a range of random numbers.

Objectives (cont.)

- Learn simulation techniques using random-number generation.
- Seed the random number generator to ensure reproducibility.
- Pack values into a tuple and unpack values from a tuple.
- Return multiple values from a function via a tuple.
- Understand how an identifier's scope determines where in your program you can use it.

Objectives (cont.)

- Create functions with default parameter values.
- Call functions with keyword arguments.
- Create functions that can receive any number of arguments.
- Use methods of an object.

Outline

4.2 Defining Functions

- `square` function that calculates the square of its argument.

In [1]:

```
def square(number):
    """Calculate the square of number."""
    return number ** 2
```

In [2]:

```
square(7)
```

Out[2]:

```
49
```

In [3]:

```
square(2.5)
```

Out[3]:

```
6.25
```

- Calling `square` with a non-numeric argument like `'hello'` causes a `TypeError` because the exponentiation operator (`**`) works only with numeric values

Defining a Custom Function

- Definition begins with the (**`def` keyword**, followed by the function name, a set of parentheses and a colon (**`:`**).
- By convention function names should begin with a lowercase letter and in multiword names underscores should separate each word.
- Required parentheses contain the function's **parameter list**.
- Empty parentheses mean no parameters.
- The indented lines after the colon (**`:`**) are the function's **block**
    - Consists of an optional docstring followed by the statements that perform the function's task.

Specifying a Custom Function's Docstring

- *Style Guide for Python Code*: First line in a function's block should be a docstring that briefly explains the function's purpose.

Returning a Result to a Function's Caller

- Function calls also can be embedded in expressions:

In [4]:

```
print('The square of 7 is', square(7))
```

```
The square of 7 is 49
```

- Three Ways to Return a Result to a Function's Caller
  - `return` followed by an expression.
  - `return` without an expression implicitly returns `None` —represents the **absence of a value** and **evaluates to `False` in conditions**.
  - **No `return` statement implicitly returns `None`.**

What Happens When You Call a Function

- Parameters exist only during the function call.
- Created on each call to the function to receive arguments.
- Destroyed when the function returns its result to the caller.
- A function's parameters and variables defined in its block are all **local variables**.

Accessing a Function's Docstring Via IPython's Help Mechanism

- Following a function's name with `?` in IPython displays its docstring:

In [5]:

```
square?
```

```
Signature: square(number)
Docstring: Calculate the square of number.
File:      ~/Dropbox/books/2019/Python/PyCDS_JupyterSlides/ch04/<ipy
thon-input-1-7d5dc51751d0>
Type:      function
```

- If the function's source code is accessible from IPython, `??` displays the function's docstring and full source-code definition:

In [6]:

```
square??
```

```
Signature: square(number)
Source:
def square(number):
    """Calculate the square of number."""
    return number ** 2
File:      ~/Dropbox/books/2019/Python/PyCDS_JupyterSlides/ch04/<ipy
thon-input-1-7d5dc51751d0>
Type:      function
```

4.4 Random-Number Generation

- Can introduce the element of chance via the Python Standard Library's `random` module.

Rolling a Six-Sided Die

- Product 10 random integers in the range 1–6 to simulate rolling a six-sided die:

In [1]:

```
import random
```

In [2]:

```
for roll in range(10):
    print(random.randrange(1, 7), end=' ')
```

```
6 2 3 5 6 5 1 1 3 2
```

- `randrange` function generates an integer from the first argument value up to, but not including, the second argument value.
- Different values are displayed if you re-execute the loop.

In [3]:

```
for roll in range(10):
    print(random.randrange(1, 7), end=' ')
```

```
3 3 4 2 6 2 4 3 5 6
```

- Can guarantee **reproducibility** of a random sequence with the `random` module's `seed` function.

Rolling a Six-Sided Die 6,000,000 Times

- If `randrange` truly produces integers at random, every number in its range has an equal probability (or chance or likelihood) of being returned each time we call it.
- Roll a die 6,000,000 times.
- Each die face should occur approximately 1,000,000 times.
- We used Python's underscore (_) digit separator to make the value 6000000 more readable.

```python
### Rolling a Six-Sided Die 6,000,000 Times

# fig04_01.py
"""Roll a six-sided die 6,000,000 times."""
import random

# face frequency counters
frequency1 = 0
frequency2 = 0
frequency3 = 0
frequency4 = 0
frequency5 = 0
frequency6 = 0

# 6,000,000 die rolls
for roll in range(6_000_000):  # note underscore separators
    face = random.randrange(1, 7)

    # increment appropriate face counter
    if face == 1:
        frequency1 += 1
    elif face == 2:
        frequency2 += 1
    elif face == 3:
        frequency3 += 1
    elif face == 4:
        frequency4 += 1
    elif face == 5:
        frequency5 += 1
    elif face == 6:
        frequency6 += 1

print(f'Face{"Frequency":>13}')
print(f'{1:>4}{frequency1:>13}')
print(f'{2:>4}{frequency2:>13}')
print(f'{3:>4}{frequency3:>13}')
print(f'{4:>4}{frequency4:>13}')
print(f'{5:>4}{frequency5:>13}')
print(f'{6:>4}{frequency6:>13}')
```

In [4]:

```
run fig04_01.py
```

```
Face    Frequency
   1      1000562
   2       999042
   3       999988
   4      1000966
   5       999281
   6      1000161
```

Seeding the Random-Number Generator for Reproducibility

- Function `randrange` generates pseudorandom numbers.
- Numbers appear to be random, because each time you start a new interactive session or execute a script that uses the random module's functions, Python internally uses a different seed value.
- When you're debugging logic errors in programs that use randomly generated data, it can be helpful to use the same sequence of random numbers.
- To do this, use the random module's `seed` function to seed the random-number generator:

In [5]:

```
random.seed(32)
```

In [6]:

```
for roll in range(10):
    print(random.randrange(1, 7), end=' ')
```

```
1 2 2 3 6 2 4 1 6 1
```

In [7]:

```
for roll in range(10):
    print(random.randrange(1, 7), end=' ')
```

```
1 3 5 3 1 5 6 4 3 5
```

In [8]:

```
random.seed(32)
```

In [9]:

```
for roll in range(10):
    print(random.randrange(1, 7), end=' ')
```

```
1 2 2 3 6 2 4 1 6 1
```

4.6 Python Standard Library

- You write Python programs by combining functions and classes (that is, custom types) that you create with preexisting functions and classes defined in modules, such as those in the Python Standard Library and other libraries.
- Avoid “reinventing the wheel.”
- A module is a file that groups related functions, data and classes.
- A **package** groups related modules.
- The Python Standard Library (https://docs.python.org/3/library/) is provided with the core Python language.
- Its packages and modules contain capabilities for a wide variety of everyday programming tasks.

Some popular Python Standard Library modules
`collections` —Data structures beyond lists, tuples, dictionaries and sets.
Cryptography modules—Encrypting data for secure transmission.
`csv` —Processing comma-separated value files (like those in Excel).
`datetime` —Date and time manipulations. Also modules `time` and `calendar` .
`decimal` —Fixed-point and floating-point arithmetic, including monetary calculations.
`doctest` —Embed validation tests and expected results in docstrings for simple unit testing.
`gettext` and `locale` —Internationalization and localization modules.
`json` —JavaScript Object Notation (JSON) processing used with web services and NoSQL document databases.
`math` —Common math constants and operations.
`os` —Interacting with the operating system.
`profile` , `pstats` , `timeit` —Performance analysis.
`random` —Pseudorandom numbers.
`re` —Regular expressions for pattern matching.
`sqlite3` —SQLite relational database access.
`statistics` —Mathematical statistics functions such as `mean` , `median` , `mode` and `variance` .
`string` —String processing.
`sys` —Command-line argument processing; standard input, standard output and standard error streams.
`tkinter` —Graphical user interfaces (GUIs) and canvas-based graphics.
`turtle` —Turtle graphics.
`webbrowser` —For conveniently displaying web pages in Python apps.

4.7 `math` Module Functions

- The **`math` module** defines functions for performing various common mathematical calculations.

In [1]:

```
import math
```

In [2]:

```
math.sqrt(900)
```

Out[2]:

```
30.0
```

In [3]:

```
math.fabs(-10)
```

Out[3]:

```
10.0
```

- Some `math` module functions are summarized below
- [View the complete list (https://docs.python.org/3/library/math.html)](https://docs.python.org/3/library/math.html)

Function	Description	Example
`ceil( x )`	Rounds x to the smallest integer not less than x	`ceil(9.2) is 10.0` , `ceil(-9.8) is -9.0`
`floor( x )`	Rounds x to the largest integer not greater than x	`floor(9.2) is 9.0` , `floor(-9.8) is -10.0`
`sin( x )`	Trigonometric sine of x (x in radians)	`sin(0.0) is 0.0`
`cos( x )`	Trigonometric cosine of x (x in radians)	`cos(0.0) is 1.0`
`tan( x )`	Trigonometric tangent of x (x in radians)	`tan(0.0) is 0.0`
`exp( x )`	Exponential function e^x	`exp(1.0) is 2.718282` , `exp(2.0) is 7.389056`
`log( x )`	Natural logarithm of x (base e)	`log(2.718282) is 1.0` , `log(7.389056) is 2.0`
`log10( x )`	Logarithm of x (base 10)	`log10(10.0) is 1.0` , `log10(100.0) is 2.0`
`pow( x, y )`	x raised to power y (x^y)	`pow(2.0, 7.0) is 128.0` , `pow(9.0, .5) is 3.0`
`sqrt( x )`	square root of x	`sqrt(900.0) is 30.0` , `sqrt(9.0) is 3.0`
`fabs( x )`	Absolute value of x—always returns a float. Python also has the built-in function `abs` , which returns an `int` or a `float` , based on its argument.	`fabs(5.1) is 5.1` , `fabs(-5.1) is 5.1`
`fmod( x, y )`	Remainder of x/y as a floating-point number	`fmod(9.8, 4.0) is 1.8`

4.9 Default Parameter Values

- You can specify that a parameter has a **default parameter value**.
- When calling the function, if you omit the argument for a parameter with a default parameter value, the default value for that parameter is automatically passed.

In [1]:

```
def rectangle_area(length=2, width=3):
    """Return a rectangle's area."""
    return length * width
```

- Specify a default parameter value by following a parameter's name with an `=` and a value.
- Any parameters with default parameter values must appear in the parameter list to the *right* of parameters that do not have defaults.

In [2]:

```
rectangle_area()
```

Out[2]:

```
6
```

In [3]:

```
rectangle_area(10)
```

Out[3]:

```
30
```

In [4]:

```
rectangle_area(10, 5)
```

Out[4]:

```
50
```

In [ ]:

```
##########################################################################
# (C) Copyright 2019 by Deitel & Associates, Inc. and                    #
# Pearson Education, Inc. All Rights Reserved.                           #
#                                                                        #
# DISCLAIMER: The authors and publisher of this book have used their     #
# best efforts in preparing the book. These efforts include the          #
# development, research, and testing of the theories and programs        #
# to determine their effectiveness. The authors and publisher make       #
# no warranty of any kind, expressed or implied, with regard to these    #
# programs or to the documentation contained in these books. The authors #
# and publisher shall not be liable in any event for incidental or       #
# consequential damages in connection with, or arising out of, the       #
# furnishing, performance, or use of these programs.                     #
##########################################################################
```

4.10 Keyword Arguments

- When calling functions, you can use **keyword arguments** to pass arguments in `any` order.

In [1]:

```
def rectangle_area(length, width):
    """Return a rectangle's area."""
    return length * width
```

- Each keyword *argument in a call* has the form *parametername=value*.
- Order of keyword arguments does not matter.

In [2]:

```
rectangle_area(width=5, length=10)
```

Out[2]:

```
50
```

In [ ]:

4.11 Arbitrary Argument Lists

- Functions with **arbitrary argument lists**, such as built-in functions `min` and `max`, can receive *any* number of arguments.
- Function `min`'s documentation states that `min` has two *required* parameters (named `arg1` and `arg2`) and an optional third parameter of the form `*args`, indicating that the function can receive any number of additional arguments.
- The `*` before the parameter name tells Python to pack any remaining arguments into a tuple that's passed to the `args` parameter.

Defining a Function with an Arbitrary Argument List

- `average` function that can receive any number of arguments.

In [1]:

```
def average(*args):
    return sum(args) / len(args)
```

- The `*args` parameter must be the *rightmost* parameter.

In [2]:

```
average(5, 10)
```

Out[2]:

```
7.5
```

In [3]:

```
average(5, 10, 15)
```

Out[3]:

```
10.0
```

In [4]:

```
average(5, 10, 15, 20)
```

Out[4]:

```
12.5
```

Passing an Iterable's Individual Elements as Function Arguments

- Can unpack a tuple's, list's or other iterable's elements to pass them as individual function arguments.
- The `*` **operator**, when applied to an iterable argument in a function call, unpacks its elements.

In [5]:

```
grades = [88, 75, 96, 55, 83]
```

In [6]:

```
average(*grades)
```

Out[6]:

```
79.4
```

- Equivalent to `average(88, 75, 96, 55, 83)`.

4.13 Scope Rules

- Each identifier has a `scope` that determines where you can use it in your program.
- [Complete scope details (https://docs.python.org/3/tutorial/classes.html#python-scopes-and-namespaces)](https://docs.python.org/3/tutorial/classes.html#python-scopes-and-namespaces).

Local Scope

- A local variable's identifier has **local scope**.

Global Scope

- Identifiers defined outside any function (or class) have **global scope**—these may include functions, variables and classes.

Accessing a Global Variable from a Function

In [1]:

```
x = 7
```

In [2]:

```
def access_global():
    print('x printed from access_global:', x)
```

In [3]:

```
access_global()
```

```
x printed from access_global: 7
```

- By default, you cannot *modify* a global variable in a function
- Python creates a **new local variable** when you first assign a value to a variable in a function's block.
- In function `try_to_modify_global`'s block, the local `x` **shadows** the global `x`, making it inaccessible in the scope of the function's block.

In [4]:

```
def try_to_modify_global():
    x = 3.5
    print('x printed from try_to_modify_global:', x)
```

In [5]:

```
try_to_modify_global()
```

```
x printed from try_to_modify_global: 3.5
```

In [6]:

```
x
```

Out[6]:

7

- To modify a global variable in a function’s block, you must use a **`global`** statement to declare that the variable is defined in the global scope:

In [7]:

```
def modify_global():
    global x;
    x = 'hello'
    print('x printed from modify_global:', x)
```

In [8]:

```
modify_global()
```

```
x printed from modify_global: hello
```

In [9]:

```
x
```

Out[9]:

```
'hello'
```

Blocks vs. Suites

- When you create a variable in a block, it’s *local* to that block.
- When you create a variable in a control statement’s suite, the variable’s scope depends on where the control statement is defined:
  - If it's in the global scope, any variables defined in the control statement have **global scope**.
  - If it's in a function’s block, any variables defined in the control statement have **local scope.**

Shadowing Functions

- In the preceding chapters, when summing values, we stored the sum in a variable named `total` .
- If you define a variable named `sum` , it *shadows* the built-in function `sum` , making it inaccessible in your code.

In [10]:

```
sum = 10 + 5
```

In [11]:

```
sum
```

Out[11]:

```
15
```

In [12]:

```
sum([10, 5])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-12-1237d97a65fb> in <module>
----> 1 sum([10, 5])

TypeError: 'int' object is not callable
```

Statements at Global Scope

- Script statements at global scope execute as soon as they're encountered by the interpreter, whereas statements in a block execute only when the function is called.

4.14 `import` : A Deeper Look

Importing Multiple Identifiers from a Module

- Use `from…import` to import a comma-separated list of identifiers from a module then use them without having to precede them with the module name and a dot (`.`):

In [1]:

```
from math import ceil, floor
```

In [2]:

```
ceil(10.3)
```

Out[2]:

```
11
```

In [3]:

```
floor(10.7)
```

Out[3]:

```
10
```

Caution: Avoid Wildcard Imports

- Import *all* identifiers defined in a module with a **wildcard `import`** .
- Makes all of the module's identifiers available.
- Can lead to subtle errors.
- Considered a dangerous practice.

In [4]:

```
e = 'hello'
```

In [5]:

```
from math import *
```

In [6]:

```
e
```

Out[6]:

```
2.718281828459045
```

- After executing the import, variable `e` is replaced, possibly by accident, with the `math` module's constant `e`.

Binding Names for Modules and Module Identifiers

- Sometimes it's helpful to import a module and use an abbreviation for it to simplify your code.
- The `import` statement's **`as`** clause allows you to specify the name used to reference the module's identifiers.

In [7]:

```
import statistics as stats
```

In [8]:

```
grades = [85, 93, 45, 87, 93]
```

In [9]:

```
stats.mean(grades)
```

Out[9]:

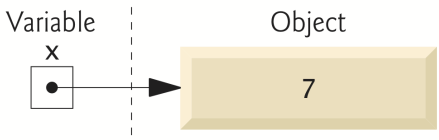
```
80.6
```

4.15 Passing Arguments to Functions: A Deeper Look

- **Python arguments are always passed by reference.**
- Some people call this **pass-by-object-reference**, because “everything in Python is an object.”
- When a function call provides an argument, Python copies the argument object’s *reference*—not the object itself—into the corresponding parameter.

Memory Addresses, References and “Pointers”

- After an assignment like the following, the variable `x` contains a reference to an *object* containing `7` stored *elsewhere* in memory.



Built-In Function id and Object Identities

- Every object has a **unique identity**—an `int` value which **identifies only that object** while it remains in memory.
- **Built-in `id` function** to obtain an object's identity.

In [1]:

```
x = 7
```

In [2]:

```
id(x)
```

Out[2]:

```
4311725472
```

Passing an Object to a Function

In [3]:

```
def cube(number):
    print('id(number):', id(number))
    return number ** 3
```

In [4]:

```
cube(x)
```

```
id(number): 4311725472
```

Out[4]:

```
343
```

- The identity displayed for `cube`'s parameter `number` is the *same* as that displayed for `x` previously.
- The *argument* `x` and the *parameter* `number` refer to the *same object* while `cube` executes.

Testing Object Identities with the is Operator

- The **`is` operator** returns `True` if its two operands have the *same identity*:

In [5]:

```
def cube(number):
    print('number is x:', number is x)  # x is a global variable
    return number ** 3
```

In [6]:

```
cube(x)
```

```
number is x: True
```

Out[6]:

```
343
```

Immutable Objects as Arguments

- When a function receives as an argument a reference to an *immutable* (unmodifiable) object, even though you have direct access to the original object in the caller, you cannot modify the original immutable object's value.

In [7]:

```
def cube(number):
    print('id(number) before modifying number:', id(number))
    number **= 3
    print('id(number) after modifying number:', id(number))
    return number
```

In [8]:

```
cube(x)
```

```
id(number) before modifying number: 4311725472
id(number) after modifying number: 4348973456
```

Out[8]:

```
343
```

In [9]:

```
print(f'x = {x}; id(x) = {id(x)}')
```

```
x = 7; id(x) = 4311725472
```

Mutable Objects as Arguments

- We'll show that when a reference to a *mutable* object like a list is passed to a function, the function *can* modify the original object in the caller.

4.18 Intro to Data Science: Measures of Dispersion

- Considered the measures of central tendency—mean, median and mode.
- Help us categorize typical values in a group.
- An entire group is called a **population.**
- Sometimes a population is quite large, such as the people likely to vote in the next U.S. presidential election, which is a number in excess of 100,000,000 people.
- For practical reasons, the polling organizations trying to predict who will become the next president work with carefully selected small subsets of the population known as **samples.**
- Hear we introduce **measures of dispersio**n (also called **measures of variability**) that help you understand how **spread out** the values are.
- We'll calculate each measure of dispersion both by hand and with functions from the module `statistics` , using the following population of 10 six-sided die rolls:

> 1, 3, 4, 2, 6, 5, 3, 4, 5, 2

Variance

- To determine variance, begin with the mean of these values—3.5.
- Next, subtract the mean from every die value:

> -2.5, -0.5, 0.5, -1.5, 2.5, 1.5, -0.5, 0.5, 1.5, -1.5

- Then, square each of these results (yielding only positives):

> 6.25, 0.25, 0.25, 2.25, 6.25, 2.25, 0.25, 0.25, 2.25, 2.25

- Finally, calculate the mean of these squares, which is 2.25 (22.5 / 10)—this is the **population variance.**
- Squaring the difference between each die value and the mean of all die values emphasizes **outliers**—the values that are farthest from the mean—which can be important in data analysis.
- The following code uses the `statistics` module's `pvariance` function to confirm our manual result:

In [1]:

```
import statistics
```

In [2]:

```
statistics.pvariance([1, 3, 4, 2, 6, 5, 3, 4, 5, 2])
```

Out[2]:

```
2.25
```

Standard Deviation

- The standard deviation is the square root of the variance (in this case, 1.5), which tones down the effect of the outliers.
- The smaller the variance and standard deviation are, the closer the data values are to the mean and the less overall dispersion (that is, spread) there is between the values and the mean.
- The following code calculates the population standard deviation with the `statistics` module's `pstdev` function, confirming our manual result:

In [3]:

```
statistics.pstdev([1, 3, 4, 2, 6, 5, 3, 4, 5, 2])
```

Out[3]:

```
1.5
```

In [4]:

```
import math
```

In [5]:

```
math.sqrt(statistics.pvariance([1, 3, 4, 2, 6, 5, 3, 4, 5, 2]))
```

Out[5]:

```
1.5
```

Advantage of Population Standard Deviation vs. Population Variance

- Suppose you've recorded the March Fahrenheit temperatures in your area.
- You might have 31 numbers such as 19, 32, 28 and 35.
- The units for these numbers are degrees.
- When you square your temperatures to calculate the population variance, the units of the population variance become **"degrees squared."**
- When you take the square root of the population variance to calculate the population standard deviation, the units once again become **degrees**, which are the same units as your temperatures.
