A Minimal Book Example

Yihui Xie

2017-12-19

Contents

Chapter 1

Prerequisites

This is a *sample* book written in **Markdown**. You can use anything that Pandoc's Markdown supports, e.g., a math equation $a^2 + b^2 = c^2$.

For now, you have to install the development versions of **bookdown** from Github:

```
devtools::install_github("rstudio/bookdown")
```

Remember each Rmd file contains one and only one chapter, and a chapter is defined by the first-level heading `#`.

To compile this example to PDF, you need to install XeLaTeX.

Chapter 2

Functions

In this module, we'll explore how to use and create **functions** in Python. Functions are the primary form of *behavior abstraction* in computer programming, and used to structure and generalize code instructions. After considering a function in an abstract sense, we'll look at how to use built-in Python functions, how to access additional functions by importing Python modules, and finally how to write our own functions.

Contents

2.1 Resources

- Functions (Sweigart)
- Functions (Severance)
- Fruitful Functions (Downey)

2.2 What are Functions?

In a broad sense, a **function** is a named sequence of instructions (lines of code) that you may want to perform one or more times throughout a program. They provide a way of *encapsulating* multiple instructions into a single "unit" that can be used in a variety of different contexts. So rather than needing to repeatedly write down all the individual instructions for "make a sandwich" every time you're hungry, you can define a `make_sandwich()` function once and then just **call** (execute) that function when you want to perform those steps.

In addition to grouping instructions, functions in programming languages like Python also follow the mathematical definition of functions, which is a set of operations (instructions!) that are performed on some **inputs** and lead to some **outputs**. Functions inputs are called **arguments** or **parameters**, and we say that these arguments are **passed** to a function (like a football). We say that a function then **returns** an ouput for us to use.

2.3 Python Function Syntax

Python functions are referred to by name (technically they are values like any other variable). As in many programming languages, we **call** a function by writing the name of the function followed immediately (no space) by parentheses `()`. Inside the parentheses, we put the **arguments** (inputs) to the function separated by commas. Thus computer functions look just like mathematical functions, but with names longer than `f()`.

```
# call the print() function, pass it "Hello world" value as an argument
print("Hello world")  # "Hello world"

# call the str() function, pass it 598 as an argument
str(598)  # "598"

# call the len() function, pass it "python" as an argument
len("python")  # 6 (the word is 6 letters long)

# call the round() function, pass it 3.1415 as the first arg and 2 as the second
# this is an example of a function that takes multiple (ordered) args
round(3.1415, 2)  # 3.14, (3.1415 rounded to 2 decimal places)

# call the min() function, pass it 1, 6/8, AND 4/3 as arguments
# this is another example of a function that takes multiple args
min(1, 6/8, 4/3)  # 0.75, (6/8 is the smallest value)
```

- *Note:* To keep functions and variables distinct, I try to always include empty parentheses `()` when referring to a function name. This does *not* mean that the function takes no arguments, it is just a useful shorthand for indicating that something is a function rather than a variable.

Some functions (such as `min()` or `print()`) can be passed as many arguments as you wish: `min()` will find the minimum of *all* the arguments, and `print()` will print *all* the arguments (in order), separated by a space:

```
# print() with 3 arguments instead of 1
print("I", "love", "programming")  # "I love programming"
```

Besides ordered **positional arguments**, functions may also take **keyword arguments**, which are arguments for specific function inputs. These are written like variable assignments, but within the function parameters:

```
# Use the `sep` keyword argument to specify the separator is '+++'
print("Hello", "World", sep='+++')  # "Hello+++World"
```

Keyword arguments are always optional (they have "default" values, like how the separator for `print()` defaults to a single space `' '`). The default values are specified in the function documentation (e.g., for `print()`).

If you call any of these functions interactively (e.g., in an interactvie shell or a Jupyter notebook), Python will display the **returned value**. However, the computer is not able to "read" what is written to the console or an output cell—that's for humans to view! If we want the computer to be able to *use* a returned value, we will need to give that value a name so that the computer can refer to it. That is, we need to store the returned value in a variable:

```
# store min value in smallest_number variable
smallest_number = min(1, 6/8, 4/3, 5+9)

# we can then use the variable as normal, such as mathematical operations
twice_min = smallest_number * 2  # 1.5
```

```
# we can use functions directly in expressions (the returned value is anonymous)
number = .5 * round(9.8)  # 5.0

# we can even pass the result of a function as an argument to another!
# watch out for where the parentheses close!
print(min(2.0, round(1.4)))  # prints 1
```

2.3.1 Object Methods

In Python, all data values are **objects**, which are groups of data (called *attributes*) and behaviors—that is, information about the value and the *functions* that can be applied to that data. For example, a Person object may have a name (e.g., `"Ada"`) and some behavior it can do to that data (e.g., `say_name()`). Functions that are applied to an object's data are also known as **methods**. We say that a method is *called on* that object.

While we'll discuss objects in more much detail later, for now we need to understand that some functions are called *on* particular values. This is done using **dot notation**: you write the name of the variable you wish to call the method on (i.e., apply the function to), followed by a period (dot) `.`, followed by the method name and arguments:

```
message = "Hello World"

# call the lower() method on the message to make a lowercase version
# Note that the original string does not change (strings are immutable)
lower_message = message.lower()  # "hello world"

# call the replace() method on the message
western_message = message.replace("Hello", "Howdy")  # "Howdy World"
```

This is a common way of utilizing built-in Python functions.

- Note that dot notation is also used to access the **attributes** or **properties** of an object. So if a `Person` object has a `name` attribute, you would refer to that as `person.name`. In this sense, you can think of the *dot operator* as being like the possessive `'s` in English: `person.name` refers so "`person`**'s** `name`'".

2.4 Built-in Python Functions

As you may have noticed, Python comes with a large number of functions that are built into the language. In the above examples, we used the `print()` function to print a value to the console, the `min()` function to find the smallest number among the arguments, and the `round()` function to round to whole numbers. Similarly, each data type in the language comes with their own set of methods: such as the `lower()` and `replace()` methods on strings.

These functions are all described in the official documentation. For example, you can find a list of built-in functions (though most of them will not be useful for us), as well as a list of string methods. To learn more about any individual function as well as what functions are available, look it up in the documentation. You can also use the `help()` function to look up the details of other functions: `help(print)` will look up information on the `print()` function.

Important "Knowing" how to program in a language is to some extent simply "knowing" what provided functions are available in that language. Thus you should look around and become familiar with these functions... but *do not* feel that you need to memorize them! It's enough to simply be aware "oh yeah, there was a function that rounded numbers", and then be able to look up the name and arguments for that function.

2.4.1 Modules and Libraries

While Python has lots of built-in functions, many of them are not immediately available for use. Instead, these functions are organized into **modules**, which are collections of related functions and variables. You must specifically load a module into your program in order to access it's functions; this helps to reduce the amount of memory that the Python interpreter needs to use in order to keep track of all of the functions available.

You load a module (make it available to your program) by using the `import` keyword, followed by the name of the module. This only needs to be done once per script execution, and so is normally done at the "top" of the script (in a Jupyter notebook, you can include an "importing" code cell, or import the module at the top of the cell in which you first need it):

```
# load the math module, which contains mathematical functions
import math

# call the math module's sqrt() function to calculate square root
math.sqrt(25)  # 5.0, (square root of 25)

# print out the math modules `pi` variable
print(math.pi)  # 3.141592653589793
```

Notice that we again use **dot notation** to call functions of a particular module. Again, think of the `.` as a possessive `'s` ("the `math` module**'s** `sqrt` function").

It is also possible to import only select functions or variables from a module by using `from MODULE import FUNCTION`. This will load the specific functions or variables into the *global scope*, allowing you to call them without specifying the module:

```
# import the sqrt() function from the math module
from math import sqrt

# call the imported sqrt() function
sqrt(25)  # 5.0

# import multiple values by separating them with commas
from math import sin, cos pi

sin(pi/2)  # 1.0
cos(pi/2)  # 6.123233995736766e-17; 0 but with precision errors

# import everything from math
# this is useful for module with long or confusing names
from math import *
```

The collection of built-in modules such as `math` make up what is called the *standard library* of Python functions. However, it's also possible to download and import additional modules written and published by the Python community—what are known as **libraries** or **packages**. Because many Python users encounter the same challenges, programmers are able to benefit from the work of other and *reuse* solutions. This is the amazing thing about the open-source community: people solve problems and then make those solutions available to others.

A large number of additional libraries are included with the Anaconda distribution you installed in module 1. Anaconda also comes with a command-line utility `conda` that can be used to easily download and install new libraries. For example if you needed to install Jupyter, you could use the command `conda install jupyter` from the command-line.

- Python also comes with a command-line utility called `pip` for ("pip installs packages") that can similarly be used to easily download and install packages. Note that you may need to set up a virtual environment to effectively use `pip`; thus I recommend you rely on `conda` instead-though the Anaconda package has everything you will need for this course.

2.5 Writing Functions

Even more common than loading other peoples' functions is writing your own. Functions are the primary way that we *abstract* program instructions, acting sort of like "mini progams" inside your larger script. As Downey says:

> Their primary purpose is to help us organize programs into chunks that match how we think about the problem.

Any time you want to organize your thinking—or if you have a task that you want to repeat through the script—it's good practice to write a function to perform that task. This will make your program easier to understand and uilt, as well as limit repetition and reduce the likelihood of error.

The best way to understand the syntax for defining a function is to look at an example:

```python
# A function named `MakeFullName` that takes two arguments
# and returns the "full name" made from them
def make_full_name(first_name, last_name):
    # Function body: perform tasks in here
    full_name = first_name + " " + last_name

    # Return: what you want the function to output
    return full_name

# Call the make_ful_name function with the values "Alice" and "Kim"
my_name = make_full_name("Alice", "Kim")  # "Alice Kim"
```

Functions have a couple of pieces to them:

- **`def`**: Functions are defined by using the `def` keyword, which indicates that you are defining a function rather than a regular variable. This is followed by the **name** of the function, then a set of parentheses containing the arguments (Note that the `function_name(arguments)` syntax mirrors how functions are called).

The line ends with a colon `:` which starts the function body (see below).

- **Arguments**: The values put betweeen the parentheses in the function definition line are variables that *will contain* the values passed in as **arguments**. For example, when we call `make_full_name("Alice","Kim")`, the value of the first argument (`"Alice"`) will be assigned to the first variable (`first_name`), and the value of the second argument (`"Kim"`) will be assigned to the second variable (`last_name`).

Importantly, we could have made the argument names anything we wanted (`name_first`, `given_name`, etc.), just as long as we then use *that variable name* to refer to the argument while inside the function. Moreover, these argument variable names *only apply* while inside the function (they are **scoped** to the function). You can think of them like "nicknames" for the values. The variables `first_name`, `last_name`, and `full_name` only exist within this particular function.

Note that a function may have no arguments, causing the parenthese to be empty:

`python   def say_hello():       print("Hello world!")`

You can give a function *keyword arguments* by assigning a default value to the argument in the function definition:

"'python # includes a single keyword argument def greet(greeting = "Hello"): print(greeting + " world")

# call by assigning to the arg greet(greeting = "Hi") # "Hi world"

# call wihout assigning an argument, using the default value greet() # "Hello world" "'

- **Body**: The body of the function is a **block** of code. The funcion definition ends with a colon `:` to indicate that it is followed by a *block*, which is a list of Python statements (lines of code). Which statements are part of the block are indicated by *indentation*: statements are indented by 4 spaces to make them part of that particular block. A block can contain as many lines of code as you want—you'll usually want more than 1 to make a function workwhile, but if you have more than 20 you might want to break your code up into separate functions. You can use the argument variables in here, create new variables, call other functions... basically any code that you would write outside of a function can be written inside of one as well!

- **Return value**: You can specify what output a function produces by using the **`return`** keyword, followed by the value that you wish *your function* to return (output). A `return` statement will end the current function and *return* the flow of code execution to whereever this function was called from. Note that even though we returned a variable called `full_name`, that variable was *local* to the function and so doesn't exist outside of it; thus we have to take the returned value and assign it to a new variable (as with `my_name = make_full_name("Alice", "Kim")`).

Because the `return` statement exits the function, it is almost always the last line of code in the function; any statements after a `return` statement will not be run!

- A function does not need to return a value (as in the `say_helo()` example above). In this case, omit the `return` statement.

We can call (execute) a function we defined the same way we called built-in functions. When we do so, Python will take the **arguments** we passed in (e.g., `"Alice"` and `"Kim"`) and assign them to the *argument variables*. Then the interpreter executes each line of code in the **function body** one at a time. When it gets to the `return` statement, it will end the function and return the given value, which can then be assigned to a different variable (outside of the function).

2.5.1 Doc Strings

We create new functions as ways of *abstracting* behavior, in order to make programs easier to read and understand. Thus it is important to be clear about the purpose and use of any functions you create. This can partially done through effective function and argument names: `def calc_rectangle_area(width, height)` is pretty self-explanatory, whereas `def func(a,b,c)` isn't).

- *Style requirement:* Name functions as phrases starting with *verbs* (because function do things), and variables as nouns.

Nevertheless, in order to make sure that functions are as clear as possible, you should include **documentation** in the form of a *comment* that describes in plain English what a function does: its inputs (arguments) once, output (return value), and overall behavior. In Python we document functions by providing a **doc string**. This is a string literal (written in multi-line triple quotes `"""`) placed immediately below the function declaration:

```
def to_celcius(degrees_farenheit):
    """Converts the given degrees (in Farenheit) to degrees in celcius, and
       returns the result.
    """
    celcius = (degrees_farenheit - 32)*(5/9)
    return celcius
```

- Note that this string isn't assigned to a variable; it is treated as a valid statement because it is immediately below the function declaration.

- You can view the doc string by calling the `help()` function with your function's name as the parameter! In fact, when you call `help()` on any built-in function, what you are viewing is that function's doc string.

Doc strings should include the following information:

1. What the function does *from the perspective of someone who would call the function.* This should be a short (1-2 sentence) summary; don't describe the individual instructions that are executed, but an *abstraction* of the code's purpose.

- If you mention variables, operations, functions, or control structures, your comment isn't at a high enough level of abstraction!

2. Descriptions of the expected arguments. Clarify any ambiguities in type (e.g., a number or a string) and units.
3. Description of the returned value (if any). Explain the meaning, type (e.g., number or string), etc.

It's good to think of doc strings as defining a "contract": you are specifying that "if you give the function this set of inputs, it will perform this behavior and give you this output.".

For simple methods you can build this information into a single sentence as in the above example. But for more complex functions, you may need to use a more complex format for your doc string. See the Google Style Guide for an example.

Chapter 3

Logic and Conditionals

Programming involves writing instructions for a computer to execute. However, what allows computer programs to be most useful is when they are able to *decide which* instructions instructions to execute based on a particular situation. This is refered to as code branching, and is used to shape the flow of control of the computer code. In this module, you will learn how to utilize **conditional statements** in order to include control flow in your Python scripts.

Contents

3.1 Resources

- Conditional Execution (Downey)
- Conditionals (Severance)
- Flow Control (Sweigart) (first half)

3.2 Booleans

In addition to the basic data types `int`, `float`, and `str`, Python supports a *logical* data type called a **Boolean** (class `bool`). A boolean represents "yes-or-no" data, and can be exactly one of two values: `True` or `False` (note the capitalization). Importantly, these **are not** the Strings `"True"` and `"False"`; boolean values are a different type!

```
type(True)    # <class 'bool'>
type("True")  # <class 'str'>
type(true)    # NameError: name 'true' is not defined
              # e.g., no variable called `true`!
```

- *Fun fact*: logical values are called "booleans" after mathematician and logician George Boole, who invented many of the rules and uses of this construction (called Boolean algebra).

Boolean values are most commonly the result of applying a **relational operator** (also called a **comparison operator**) to some other data type. Comparison operators are used to compare values and include: `<` (less than), `>` (greater than), `<=` (less-than-or-equal, written as read), `>=` (greater-than-or-equal, written as read), `==` (equal), and `!=` (not-equal).

```
x = 3
y = 3.15

# compare numbers
x > y  # returns logical value False ("x is bigger than y" is a False statement)
y != x  # returns logical value True ("y is not-equal to x" is a True statement)

# compare x to pi (built-in variable)
y == math.pi  # returns logical value False

# compare strings (based on alphabetical ordering)
"cat" > "dog"  # returns False
```

- **Important** `==` (two equals signs) is a comparison operator, but `=` (one equals sign) is the assignment operator!

Note that boolean variables should be named as *statements of truth*. Use words such as `is` in the variable name:

```
is_early = True
is_sleeping = False
needs_coffee = True
```

3.2.1 Boolean Operators

In addition, boolean values support their own operators (called **logical operators** or **boolean operators**). These operators are applied to boolean values and produce boolean values, and allow you to make more complex *boolean expressions*:

- `and` (conjunction) produces `True` if both of the operands are `True`, and `False` otherwise
- `or` (disjunction) produces `True` if *either* of the operands are `True`, and `False` otherwise
- `not` (negation) is a unary operator that produces `True` if the operand is `False`, and `False` otherwise

```
x = 3.1
y = 3.2

# Assign bool values to variables
x_less_than_pi = x < math.pi  # True
y_less_than_pi = y < math.pi  # False

# boolean operators
x_less_than_pi and y_less_than_pi  # False
x_less_than_pi or y_less_than_pi  # True

# this works because Python is amazing
x < math.pi < y  # True


pet = "dog"

# it is NOT the case that pet is "cat"
not pet == "cat"  # True
```

```
# pet is "cat" OR "dog"
pet == "cat" or pet == "dog"  # True

# this doesn't work (operators are applied left-to-right; check the types!)
# see "short-circuiting" below, as well as http://stackoverflow.com/a/19213583
pet == "cat" or "dog"  # "dog"
```

Because boolean expressions produce *more* booleans, it is possible to combine these into complex logical expressions:

```
# given two booleans P and Q
P = True
Q = False

P and not Q  # True
not P and Q  # False
not (P and Q)  # True
(not P) or (not Q)  # True
```

The last two expressions in the above example are equivalent logical statements for *any* combination of values for `P` and `Q`, what is known as De Morgan's Laws. Indeed, many logical statements can be written in multiple equivalent ways.

Finally, note that when using an `and` operator, Python will **short-circuit** the second operand and never evaluate it if the first is found to be `False` (after all, if the first operand isn't `True` there is no way for the entire `and` expression to be!)

```
x = 2
y = 0
x == 2 and x/y > 1  # ZeroDivisionError: division by zero
x == 3 and x/y > 1  # no error (short-circuited)

# Use a "guardian expression" (make sure y is not 0)
# to avoid any errors
y != 0 and x/y > 1  # no error (short-circuited)
```

- The reason this works is because Python interpreter reads the expression `P and Q`, it produces `Q` if `P` is `True`, and `P` otherwise. This is because if `P` is `True`, then the overall truth of the expression is dependent entirely on `Q` (and thus that can just be returned). Similarly, if `P` is `False`, then the whole statement is false (equivalent to `P`!)

3.3 Conditional Statements

One of the primary uses of Boolean values (and the *boolean expressions* that produce them) is to control the flow of execution in a program (e.g., what lines of code get run in what order). While we can use *functions* are able to organization instructions, we can also have our program decide which set of instructions to execute based on a set of conditions. These deciisions are specified using **conditional statements**.

In an abstract sense, an conditional statement is saying:

```
IF something is true
  do some lines of code
OTHERWISE
  do some other lines of code
```

In Python, we write these conditional statements using the keywords `if` and `else` and the following syntax:

```
if condition:
    # lines of code to run if condition is True
else:
    # lines of code to run if condition is False
```

The **`condition`** can be any Boolean value (or any expression that evaluates to a boolean value). Both the **`if`** statement and **`else`** clause are followed by a colon **`:`** and a **block**, which is a set of *indented* statements to run (similar to the blocks used in functions). It is also possible to *omit* the **`else`** statement and its block if there are no instructions to run in the "otherwise" situation:

```
porridge_temp = 115  # temperature in degrees F

if porridge_temp > 120:
    print("This porridge is too hot!")
else:
    print("This porridge is NOT too hot!")

too_cold = porridge_temp < 70  # a boolean variable
if too_cold:
  print("This porridge is too cold!")

# This line is outside the block, so is not part of the conditional
# (it will always be executed)
print("Time for a nap!")
```

Blocks can themselves contain *nested* conditional statements, allowing for more complex decision making. Nested **`if`** statements are indented twice (8 spaces or 2 tabs). There is no limit to how many "levels" you can nest; just increase the indentation each time.

```
# nesting example
if outside_temperature < 60:
    print("Wear a jacket")
else:
    if outside_temperature > 90:
        print("Wear sunscreen")
    else:
        if outside_temperature == 72:
            print("Perfect weather!")
        else:
            print("Wear a t-shirt")
```

Note that this form is nesting is also how we you can use conditionals inside of functions:

```
def flip(coin_is_heads):
    if coin_is_heads:
        print("Heads you win!")
    else:
        print("Tails you lose")
```

If you consider the above nesting example's logic carefully, you'll notice that many of the "branches" are **mutually exclusive**: that is, the code will choose only 1 of 4 different clothing suggestions to print. This can be written in a cleaner format by using an **`elif`** ("else if") clause:

```
if outside_temperature < 60:
    print("Wear a jacket")
elif outside_temperature > 90:
    print("Wear sunscreen")
```

```python
elif outside_temperature == 72:
    print("Perfect weather!")
else:
    print("Wear a t-shirt")
```

In this situation, the Python interpreter will perform the following logic:

1. It first checks the `if` statement's condition. If that is `True`, then that branch is executed and the rest of the clauses are skipped.
2. It then checks each `elif` clause's condition *in order*. If one of them is `True`, then that branch is executed and the rest of the clauses are skipped.
3. If *none* of the `elif` clauses are `True`, then (and only then!) the `else` block is executed.

This *ordering* is important, particularly if the conditions are not in fact mutually exclusive:

```python
if porridge_temp < 120:
    print("This porridge is not too hot!")
elif porridge_temp < 70:
  # unreachable statement! the condition will never be both checked and True
    print("This porridge is too cold!")

# contrast with:
if porridge_temp < 120:
    print("This porridge is not too hot!")
if porridge_temp < 70:  # a second if statement, unrelated to the first
    print("This porridge is too cold!")
# both print statements will execute for `porridge_temp = 50`
```

See also the resources listed at the top of the module for explanations with logical diagrams and flowcharts.

3.3.1 Designing Conditions

Relational operators all have logical opposites (e.g., `==` is the opposite of `!=`; `<=` is the opposite of `>`), and boolean expressions can include negation. This means that there are many different ways to write conditional statements that are *logically equivalent*:

```python
# these two statements are equivalent
if x > 3:
    print("big X!")

if 3 < x:
    print("big X!")
```

- The second example is known as a Yoda condition; in Python you should use the former.

Thus you should follow the below guidelines when writing conditionals. These produce more "idiomatic" code which is cleaner and easier to read, as well as less likely to cause errors.

- Avoid checks for mutually exclusive conditions with an `if` and `elif`. Use the `else` clause instead!

"'python # Do not do this! if temperature < 50: print("It is cold") elif temperature >= 50: # unnecessary condition print("It is not cold")

# Do this instead! if temperature < 50: print("It is cold") else: print("It is not cold") "'

- Avoid creating redundant boolean expressions by comparing boolean values to `True`. Instead, use an effectively named variable.

"'python # Do not do this! if is_raining == True: # unnecessary comparison print("It is raining!")

# Do this instead! if is_raining: # condition is already a boolean! print("It is raining!") "'

Note that this gets trickier when trying to check for `False` values. Consider the following equivalent conditions:

"'python # I believe this is the cleanest option, as it reads closet to English if not is_raining: print("It is not raining!")

# This is an acceptable equivalent, but prefer the first option if is_raining == False print("It is not raining!")

# Use one of the above options instead if is_raining != True print("It is not raining!")

# This can be confusing unless your logic is explicitly based around the # ABSENCE of some condition if is_not_raining: print("It is not raining!") "'

Overall, try to develop the simplest, most straightforward conditions you can. This will make sure that you are able to think clearly about the program's control flow, and help to clarify your own thinking about the program's logic.

3.3.2 Modules vs. Scripts

As discussed in module5, it is possible to define Python variables and functoins in `.py` files, which can be run as stand-alone scripts. However, these files can also be imported as modules, allowing you to access their variables and functions from another script. Thus all Python scripts have two "modes": they can be used as executable scripts (run with the `python` command at the command-line), or they can be imported as modules (libraries of variables and functions, using the `import` keyword in the script).

When the `.py` script is run as an executable top-level script, we often want to perform special instructions (e.g., call specific functions, prompt for user input, etc). We can use a special *environmental* variable called `__name__` to determine whether this is the "main" script that is being run, and `if` so execute those instructions:

```python
if __name__ == "__main__":
    # execute only if run as a script
    print("This is run as a script, not a module!")
```

- The `__` in the variable names and values are a special naming convention used to indicate *to the human* that these values are **internal** to the Python language and so have special meaning. They are otherwise normal variables names

Chapter 4

Iteration and Loops

One of the main benefits of using computers to perform tasks is that computers never get tired or bored, and so can do the same thing over and over and over and over and over and over again. This is a process known as **iteration**. Iteration represents another form of *control flow* (similar to conditionals), and is specified in a program using a set of statements called **loops**. In this module, you will learn the basics of writing loops to perform iteration; more advanced iteration concepts will be covered in later modules.

Contents

4.1 Resources

- Iterations (Downey)
- Flow Control (Sweigart) (second half)
- Iteration (Severance)
- Files (Downey)
- Files and File Paths (Sweigart)

4.2 While Loops

Loops are **control structures** (similar to `if` statements) that allow you to perform iteration. These statements specify that a *block* of code should be executed repeatedly—the block is executed statement by statement, and then the flow of control "loops" back to the top to execute the statements again. Programming languages such as Python support a number of different kinds of loops, which differ primarily in how they determine whether or not to repeat the block (though this difference is reflected in the syntax).

In programming, the most basic *control structure* used for iteration is known as a **while loop**, which is used for "indefinite iteration" A Python while loop has the following structure:

```
while condition:
    # lines of code to run if the condition is True
```

This construction looks much like an `if` statement, and is similar in many regards. As with an `if` statement, the **condition** can be any Boolean value (or any expression that evaluates to a boolean value), and it determines whether or not the loop's *block* will be executed.

In order to understand how a while loop influences the flow of program control, consider a more concrete example:

```
count = 5
while count > 0:
    print(count)
    count = count - 1

print("Blastoff!")
```

In this example, when the interpreter reaches the `while` statement, it first checks the condition (`count > 0`, where `count` is 5). Finding that condition to be `True`, the interpreter then executes the block, printing the `count` and then *decrementing* it. Once the block is executed, the interpreter loops back to the `while` statement and rechecks the condition. Finding that `count` (4) is still greater than 0, it executes the block again (causing `count` to decrement again). This continues until the interpreter loops to the `while` statement and finds that `count` has reached 0, and thus is no longer greater than 0. Since the condition is now `False`, the interpreter does *not* enter the loop, and proceeds to the following statement ("Blastoff!").

Importantly, the condition is only checked when the block is *about* to execute: both at the "start" of the loop, and then at the beginning of each subsequent iteration. Having the condition become `True` in the middle of the block (e.g., temporarily) will have no impact on the control flow. It is also possible for the interpreter to "never enter" the loop if the condition is not initially `True`.

4.2.1 Counting and Loops

The above example also demonstrates how to use a "counter" to determine whether or not the loop has run a sufficient number of times: this is known as a **loop control variable (LCV)**. The "standard" counting loop looks like:

```
count = 0  # 1. initialize the counter
while count < 100:  # 2. check if the counter has reached its target
    print(count)  # 3. do some work (this may be multiple statements)
    count = count + 1  # 4. update the counter
```

In order for a counted loop to work properly, you need to be careful about steps 2 and 4: the condition and the update.

First, recall that the condition is *whether to run the loop*, not *whether to stop*:

```
count = 0
while count == 100:  # bad condition!
    print(count)
    count = count + 1
```

In this case, the condition is not initially True, so the interpreter never enters the loop.

- When writing conditions, think *"do we keep going"* rather than *"are we there yet?"*. In loops (as in life), the journey is more important than the destination!

Second, consider what happens if you forget to update the loop control variable:

```
count = 0
while count < 100
    print(count)
    # no counter update
```

In this case, the interpreter checks that `count` (0) is less than 100, then runs the loop. Then checks that `count` (still 0) is less than 100, then runs the loop. Then checks that `count` (*still* 0) is less than 100, then runs the loop…

This is known as an **infinite loop**: the loop will run forever, never being able to "break out" and reach the next statement.

- If you hit an infinite loop in Jupyter Notebook, use `Kernel > Interrupt` to break it and try again. If running a `.py` file on the command-line, use `Ctrl-C` to cancel the script.

There are lots of ways to accidently produce an infinite loop:

1. Having a condition that is "too exact" can cause the loop control variable to "miss" a particular breaking value:

`python    count = 0    while count != 100:   # if we aren't yet at 100         print(count) count = count + 3   # this will never equal 100`

Thus it is always safe to use **inequalities** (e.g., `<` or `>`) when writing loop conditions.

2. Resetting the counter in the body of the loop can cause it to never reach its goal:

`python    count = 0    while count < 100:         count = 0   # this resets the count! print(count)       count = count + 1`

The best way to catch these errors is to "play computer": pretend that you are the compiler, and go through each statement one by one, keeping track of the loop control variable (writing its value down on a sheet of paper does wonders). This will help you be able to "trace" what your program is doing and catch any bugs there may be.

4.2.2 Conditionals and Sentinels

As with `if` statements the block for a `while` loop can contain any valid Python statements, including `if` statements or even other loops (called a "nested loop"). Control statements such as `if` are intended an extra step (4 spaces or 1 tab):

```
# flip a coin until it shows up heads
still_flipping = True
while still_flipping:
    flip = random.randint(0,1)
    if flip == 0:
        flip = "Heads"
    else:
        flip = "Tails"
    print(flip)
    if flip == "Heads":
        still_flipping = False
```

In this example, the `still_flipping` boolean variable acts as the *loop control variable*, as it determines whether or not the loop is repeated. Using a Boolean as a LCV is known as using a **sentinel variable**. A sentinel (guard) variable is used to control whether or not the program flow gets out of the loop: as long as the sentinel is `True`, the loop continues to run. Thus the loop can be "exited" by assigning the sentinel to be `False`. This is particularly useful when there may be a complex set of conditions that need to be met before the program can carry on.

- It is of course possible to design a sentinel such as `done_flipping`, and then have the `while` condition check that the sentinel is **not** `True`. This may be useful depending on how you've structured the algorithm. In either case, be sure that your sentinels are named carefully and accurately reflect the information conveyed by the variable!

4.3 For Loops

If you look back at the basic counting loop, you'll notice that tracking the `count` loop control variable can be problematic. It is easy to forget the update statement (`count = count + 1`), and the the `count` variable itself acts as an extra "global" (or "less local") variable that the interpreter needs to keep track of.

To avoid these problems, we can instead use a different kind of loop called a **for loop**. A for loop is used for "definite iteration", when we want execute a loop a specific number of times. The basic Python for loop has the following structure:

```
for local_variable in range(maximum):
    # lines of code to run for each number
```

For example, the basic counting while loop example could be rewritten using a simpler for loop:

```
for count in range(100):
  print(count)
```

`range()` is a function that returns a value representing a sequence of numbers. It is an example of a **sequence** data structure, which is a way of representing multiple data items in a single variable. Sequences will be discussed more in later modules (including the most common type of sequence, a **list**). The `range()` function can be called with different arguments depending on the range and spread of numbers you wish to use:

```
# numbers 0 to 10 (not inclusive)
# (0 through 9)
range(10)

# numbers 1 through 11 (not inclusive)
# (1 through 10)
range(1,11)

# numbers 0 through 10 (not inclusive), skipping by 3
# (0, 3, 6, 9)
range(0, 10, 3)
```

Thus `for count in range(100)` can be read as "for *each* number (called count) from 0 to 100".

For loops may more properly be thought of as **"for each"** loops; they are used to go through the items in a collection (e.g., each number in a `range`), executing the loop body once for each item. The `local_variable` (e.g., `count`) in a for loop is *implicitly* assigned the value of the "current" item in the collection (e.g., which number in the range we're on) at each iteration.

- For ranges, this basically means that the for loops keeps track of the current iteration; but the idea of this local variable will be more important as we introduce additional collection types.

4.3.1 Difference from While Loops

The main difference between while loops and for loops is:

while loops are used for **indefinite** iteration; for loops are used for **definite** iteration.

While loops are appropriate when the interpreter doesn't know *in advance* (before the loop starts) how many times the loop block will be executed: the loop does not have a definite number of iterations. On the other hand, a for loop is appropriate when the interpreter does know *in advance* (before the loop starts) how many times the block will be executed—a definite number of iterations. Note that that number of iterations may be a variable so not determined until runtime; however, the value of that variable will still be known when the `for` statement is executed.

All iteration can be written with a while loop; but when performing definite iteration, it is easier, faster, and more idiomatic to use a for loop!

4.4 Working with Files

For loops can be used to iterate through any collection (technically any "iterable" type). One of the more useful collections when working with data is external **files** (e.g., text files). Files can be treated as a collection or sequence of *lines* (each divided by a `\n` newline character), and thus Python can "read" a text file using a for loop to iterate over the lines of text in the file.

In order to read or write text file data, you use the built-in `open()` function, passing it the *path* to the file you wish to access. This function will then return an object representing that particular file (e.g., it's location on the disk), with methods that you can use to read from and write to it.

- Remember to **always** use *relative* paths. Note that when using a Jupyter Notebook, the "current working directory" is the direction in which you ran the `jupyter notebook` command to start the server.

```
my_file = open('myfile.txt')  # open the file

for line in my_file:
    print(line)  # print each line in the file
```

Once you have opened a file, you can use a for loop to iterate through its line (as in the example above). You can also use a while loop, calling the `readline()` method *on* the file in order to read a single line at a time.

It is also possible to write out content to a file. To do this, you need to open the file with "write" access (allowing the program to write to and modify it) by passing `w` as the second argument to the `open()` function. You can then use the `write()` method to "print" text to the file:

```
# "open" the file with "write" access
file = open('myfile.txt', 'w')

file.write("Hello world!\n")
file.write("It's a mighty fine morning\n")
```

- Note that unlike the `print()` function, `write()` does not include a line break at the end of each method call; you need to add those yourself!

4.4.1 Try/Except

File operations rely on a context that is internal to the program itself: namely, that the file you wish to open actually exists at the location you specify! But that may not be the case—particularly if which file to open is specified by the user:

```
filename = input("File to open: ")  # which file to open

file = open(filename)  # "open" the file
```

```
for line in file:
    print(line)
```

If the user provides a bad file name, your program will encounter an error *through no fault of your own as a programmer*:

```
$ python script.py
File to open: neener neener
Traceback (most recent call last):
  File "script.py", line 4, in <module>
    file = open(filename)
FileNotFoundError: [Errno 2] No such file or directory: 'neener neener'
```

Since it's possible for the user to make a mistake, we could like the program not to simply fail with an error, but to instead be able to "recover" and keep running (e.g., by asking the user for a different file name). We can peform this kind of **error handling** by utilizing a **`try`** statement with an **`except`** clause:

```
filename = input("File to open: ")

try: # this might break (not our fault)
    file = open(filename)
    for line in file:
        print(line)
except:  # catching FileNotFoundError
    print("No such file")
```

A `try` statement acts somewhat similar to an `if` statement; however, the `try` statement checks to see if any errors occur *within its block*. If such an error occurs, rather than the program catching, the interpreter will *immediately* jump to the `except` class and execute that block, before continuing on with the rest of the program.

- Note that this is an exception to the general rule that conditions are only checked at the start of a block; a `try` block effectively tells the computer to keep an eye out for any errors ("try this, but it might break"), with the `except` clause specifying what to do if such an error occurs.

`try` statements are used when a program may hit an error that is *not caused by programmer's code, but by an external input* (e.g., from a user or a file). You should not use a `try` statement to fix broken program logic or invalid syntax: instead, you should fix those problems directly!

Chapter 5

Lists and Sequences

In this module, we will cover using **lists** in Python, which is a data type representing a *sequence* of data values (similar to how a `string` is a squence of letters, and a `range` is a sequence of numbers). A list is a fundamental data type in Python, and key to writing almost all practical programs. Lists are used to store and organize large sets of data (and computer programs usually deal with *lots* of data). This module cover how to create, access, and utilize lists to automate the processing of larger amounts of data.

Contents

5.1 Resources

- Lists (Severance)
- Lists (Sweigart)
- Tuples (Downey)
- Sequence Types (Python Docs)

5.2 Lists

A list is a **mutable**, **ordered** sequence of values that are all stored in a single variable. For example, you can make a vector `names` that contains the strings "Sarah", "Amit", and "Zhang", or a vector `one_to_hundred` that stores the numbers from 1 to 100. Each value in a list is refered to as an **element** of that list; thus our `names` list would have 3 elements: `"Sarah"`, `"Amit"`, and `"Zhang"`.

Lists are written as literals inside *square brackets* (`[]`), with each element in the list separated by a *comma* (`,`):

```
# a list of names
names = ["Sarah", "Amit", "Zhang"]

# a list of numbers (can contain "duplicate" values)
```

```
numbers = [1, 2, 2, 3, 5, 8]

# lists can contain different types (including other lists!)
things = ["raindrops", 2, True, [5, 9, 8]]

# lists can be empty (with no elements)
empty = []
```

- List variables should be named using **plurals** (name_s_, number_s_, etc.), because lists hold multiple values!

Other sequences (such as strings or ranges) can be converted into lists by using the built-in `list()` function:

```
list("hello")  # ['h', 'e', 'l', 'l', 'o']
list(range(1,5))  # [1, 2, 3, 4]
```

5.2.1 List Indices

We can refer to individual elements in a list by their **index**, which is the number of their position in the list. Lists are *zero-indexed*, which means that positions are counted starting at `0`. For example, in the list:

```
vowels = ['a', 'e', 'i', 'o', 'u']
```

The `'a'` (the first element) is at *index* 0, `'e'` (the second element) is at index 2, and so on.

- This also means that the last element can be found at the index `length_of_list - 1`. This pattern is why values like `range(5)` *include* `0` but *exclude* the last `5`.

You can retrieve an element from a list using **bracket notation**: you refer to the element at a particular index of a list by writing the name of the list, followed by square brackets (`[]`) that contain the index of interest:

```
names = ["Sarah", "Amit", "Zhang"]

# access the element at index 0
name_first = names[0]
print(name_first)  # Sarah

# access the element at index 2
name_third = names[2]
print(name_third)  # Zhang

# accessing an index not in the list will give an error
name_fourth = names[3]  # IndexError!

# negative indices count backwards from the end
name_last = names[-1]  # Zhang
name_second_to_last = names[-2]  #Amit
```

The value inside the square brackets can any expression that resolves to an integer, including variables:

```
names[1+1]  # "Amit"

last_index = len(names) - 1  # last index is length of list - 1
names[last_index]  # "Zhang"

# Don't forget to subtract one from the length!
```

```
names[len(names)]  # IndexError!
# Using an index of `-1` is a better solution
```

It is possible to select multiple, *consecutive* elements from a list by specifying a **slice**. A slice is written as the starting and ending indices separated by a colon (`:`); the starting index is included and the ending index is *excluded*. For example:

```
letters = ['a', 'b', 'c', 'd', 'e', 'f']

# indices 1 through 3 (non-inclusive)
letters[1:3]  # ['b', 'c']

# indices 3 to the end (inclusive)
letters[3:]  # ['d', 'e', 'f']

# indices up to 3 (non-inclusive)
letters[:3]  # ['a', 'b', 'c']

# indices 2 to (2 from end) (non-inclusive)
letters[2:-2]  # ['c', 'd'], the `e` is excluded

# all the indices. This produces a new list with the same contents!
letters[:]
```

An indexed reference to a list element (e.g., `names[0]`) is effectively a *variable in its own right*: you can think of `names[0]`, `names[1]`, and `names[2]` as being equivalent to having variables `names_0`, `names_1`, `names_2` (each of which has its own value). Lists effectively provide a "shortcut" for having lots and lots of variables that are all related; instead we can "collect" those variables into a list!

Because list references are variables, they can be used anywhere that a "normal" variable can be. In particular, this means that variables can be assigned to them, allowing the list to be **mutated** (changed):

```
# a list of school supplies
school_supplies = ["Backpack", "Laptop", "Pen"]

# replace "Pen" with "Pencil"
school_supplies[2] = "Pencil"

print(school_supplies)  # ['Backpack', 'Laptop', 'Pencil']

# You can only assign values to "variables" that exist in the list!
school_supplies[3] = "Paper"  # IndexError!
```

Just as with variables: if the list index (e.g., `my_list[index]`) is on the *left* side of an assignment, it means the **variable** (which "slot" in the list). If it is on the *right* side of an assignment, it means the **value** (which element is in that slot).

Finally, note that **strings** are also *indexed sequences* of characters. Thus you can use *bracket notation* to refer to individual letters, and many string methods utilize the index in their arguments:

```
message_str = "Hello world"
message_str[1:5]  # "ello"

# find the index of the 'w'
message_str.find('w')  # 6
```

5.3 List Operations and Methods

Lists support a number of different operations and methods (functions):

```
# Addition (+) to combine lists
['a','b'] + [1,2]  # ['a', 'b', 1, 2]

# Multiplication (*) performs multiple additions
[1,2] * 3  # [1, 2, 1, 2, 1, 2]

# A sample list
s = ['a', 'b', 'c', 'd']

# Add value to end of list
s.append('x')  # add 'x' at end

# Add value in middle of list
s.insert(2, 'y')  # put 'y' at index 2 (everyone else shifts over)

# Remove from end of list ("pop off")
s.pop()  # removes and returns the last item (`x`)

# Remove from middle
s.pop(2)  # removes and returns item at index 2 (`y`)

# Remove specific value
s.remove('c')  # remove the first 'c' in the list; nothing returned

# Remove all elements
s.clear()
```

- Note that list methods such as `append()` or `clear()` usually **mutate** the existing list value and then return `None`. In comparison, string methods (such as `lower()` or `replace()`) will return a *different* string value. This is because lists can be changed, but strings cannot (they are *immutable*).

When comparing lists using a *relational operator* (e.g., `==` or `>`), the operation is applied to the lists **member-wise**: each element in the first list operand is compared to the element *at the same index* in the second list operand. If the comparison is `True` for *each* pair of elements, then the expression is `True`. In practice, this means that (a) you can use `==` to compare the contents of lists, and (b) a list is "smaller" than another if it's first item is smaller than the other's.

But be careful: just because two lists have the same contents (are `==`) does not mean that they are *the same list*! In particular, two lists can be *different objects* (values) but still have the same contents. In Python, you can test whether two values are actually *the same value* (as opposed to having the same content) using the `is` operator.

```
# With strings, literals are shared (because they cannot be mutated)
str_a = "banana"  # `a` labels string literal "banana"
str_b = "banana"  # `b` labels string literal "banana"

# Both variables label the same (literal) value
str_a is str_b  # True

# With lists, each list created is a different object!
list_a = [1,2,3]  # `a` labels a new list [1,2,3]
list_b = [1,2,3]  # `b` labels a new list [1,2,3]
```

```
a == b  # True, have same values as contents
a is b  # False, are two different objects

list_c = list_a  # `c` labels the value that `a` labels
a is c  # True, both are the same object

# Modify the list!
list_a[0] = 10
print(list_b[0])  # 1 (a different list)
print(list_c[0])  # 10 (the same list)
```

Keeping track of whether a *new* list has been created is particularly important when using lists as **arguments to functions**. Function arguments are local variables that are *assigned* the passed value—if this value is a list, then the assigned variable will refer to the same list, and any modifications to the argument will affect the value outside of the function:

```
# A version of a function that modifies the list
def delete_first(a_list):
    a_list.pop(0)  # modifies the given value

letters = ['a','b','c']
delete_first(letters)  # call function
print(letters)  # ['b', 'c'], variable is changed

# A version of a function that does not modify the list
def delete_first(a_list):
    a_list = a_list[1:]  # create new local variable (replacing old local var)

letters = ['a','b','c']
delete_first(letters)  # call function
print(letters) #=> ['a', 'b', 'c'], variable is not changed
```

5.3.1 Lists and Loops

As with other iterable sequences like strings and files, it is possible to iterate through the contents of a list by using a *for loop*:

```
numbers = [3.98, 8, 10.8, 3.27, 5.21]

for element in numbers:  # `number` is a better local variable name
    print(element)

# This will not let you modify the list, since the "number" variable is local
for number in numbers:
  number = round(number, 1)

print(numbers)  # [3.98, 8, 10.8, 3.27, 5.21], not rounded
```

In order to *modify* the list while iterating through it, you need a way to refer to the element you want to change. Since we refer to elements by their *index*, a better solution is to instead iteate through the *range of indices* rather than through the elements themselves:

```
numbers = [3.98, 8, 10.8, 3.27, 5.21]
```

```
for i in range(len(numbers)):  # `i` for "index"
    print(numbers[i])  # refer to elements by index

# Can now modify the list
for i in range(len(numbers)):
  numbers[i] = round(numbers[i], 1)  # change value at index to rounded version

print(numbers)  # [4.0, 8, 10.8, 3.3, 5.2], rounded!
```

- Note that this process of applying some change to each element in a list is known as a **mapping** (each value "maps" or goes to some transformed version of itself). We will discuss mapping more in a later module.

5.4 Nested Lists

As noted at the start of the module, lists elements can be of **any** data type (and any *combination* of data types)—including other lists! These "lists of lists" are known as **nested lists** or **2-dimensional lists** (or *3d-list* for a "list of lists of lists", etc). Nested lists are most commonly used to represent information such as *tables* or *matrices*.

Nested lists work exactly like normal lists; the elements just happen to themselves be indexable (like strings!):

```
# a list of different dinners available at a fancy party
# this list has 4 elements, each of which is a list of 3 elements
# the indentation is just for human readability
dinner_options = [
    ["chicken", "mashed potatoes", "mixed veggies"],
    ["steak", "seasoned potatoes", "asparagus"],
    ["fish", "seasoned rice", "green beans"],
    ["portobello steak", "seasoned rice", "green beans"]
]

len(dinner_options)   # 4
fish_option = dinner_options[2]  # ["fish", "seasoned rice", "green beans"]

# because fish_option is a list, we can reference its elements by index
print(fish_option[0])  # "fish"
```

In this example `fish_option` is a variable that refers to a list, and thus its elements can be accessed by index using bracket notation. But as with any operator or function, it is also possible to use bracket notation on an *anonymous value* (e.g., a literal value that has not been assigned to a variable). That is, because `dinner_options[2]` is a list, we can use bracket notation refer to an element of that list without assigning it to a variable first:

```
# Access the 2th element's 0th element
dinner_options[2][0]  # "fish"
```

This "pair of brackets" notation allows you to easily access elements within nested lists. This is particularly useful for 2d-lists that represent *tables* as a list of "rows" (often data records), each of which is a list of "column cells" (often data features):

```
# a simple table of values
table = [ ['aa','ab','ac','ad'],
          ['ba','bb','bc','bd'],
          ['ca','cb','cc','cd'] ]
```

```
row = 1  # cells starting with 'b'
col = 3  # cells ending with 'd'
table[row][col]  # "bd", the cell at row/col
```

Note that we often use *nested for loops* to iterate through a *nested list*:

```
for i in range(len(table)):  # go through each row (with index)
    for j in range(len(table[i]))  # go through each col of that row (with index)
        print(table[i][j])  # access ith row, jth column
```

- We use a `j` for the index of the nested loop, because the `i` for "index" was already taken! `row` and `col` are also excellent local variable names.

5.5 Tuples

While lists are *mutable* (changeable) sequences of data, **tuples** represent ***immutable*** sequences of data. These are useful if you want to enforce that a data value won't be changed, such as for a function argument (or a dictionary key; see module 10). Indeed, many built-in Python functions utilize tuples.

Tuples are written as *comma-separated sequences* as values. They are often placed inside parentheses for clarity (to help indicate the start and end of the tuple values):

```
letters_tuple = ('a', 'b', 'c')
print(letters_tuple)  # ('a', 'b', 'c')

# also a tuple (without parentheses)
numbers_tuple = 1, 2, 3
print(numbers_tuple)  # (1, 2, 3)

# A tuple representing a person's name, age, and whether they are hungry
# Tuple values have _implied_ meanings, which should be explained in comments
hungry_person = ('Ada', 28, True)

# In English, tuples may be named based on the number of elements
triple = (1,2,3)
double = (4,5)
single = (6,)  # extra comma indicates is a tuple, not just int `6`
empty = ()  # an empty expression is a tuple!
type(())  # <class 'tuple'> (type of empty expression)
```

- It's important to note that while we often write tuples in parentheses (and they are printed in parentheses), it is the **commas** that makes a sequence of literals into a tuple. The parentheses act just like like they do in mathematical expressions—they are only necessary to clarify ambiguity in the order-of-operations. You will find that some "idiomatic" expressions using tuples forgo the parentheses, making the syntax look more magical than it is!

Elements in tuples can be accessed by **index** using **bracket notation**, just like the elements in lists. However, tuples *cannot be modified*, so you cannot assign a new value to an index in a tuple:

```
letter_triple = ('a','b','c')
print(letter_triple[0])  # 'a'
print(letter_triple[1:3])  # ('b','c'), a tuple

letter_triple[0] = 'z'  # TypeError!
```

Tuples can be compared using *relational operators* just like lists, and have the "member-wise" comparison behavior described above. This makes it easy to order the immutable tuples just like you would order numbers or strings.

Finally, tuples provide one additional useful feature. The Python interpreter uses tuples to perform **multiple assignments**, where you assign multiple values to multiple variables in a single statement. We have already been able to assign multiple, comma-separated values to a single *tuple* variable (a process called **packing**). But Python also supports having a single sequential value (e.g., a *tuple*) be assigned to multiple, comma-separate variables (a process called **unpacking**)!

```
triple = 1, 2, 3  # assign multiple values to single variable (packing)
print(triple)  # (1, 2, 3)


x, y, z = (1,2,3)  # assign single tuple value to multiple variables (unpacking)
print(x)  # 1
print(y)  # 2
print(z)  # 3

# the VALUE in this statement is evaluated as a tuple, and then is assigned to
# multiple variables!
a, b, c = x, y, z  # a=x; b=y; c=z

# the same process can be used to swap values!
a, b = b, a
```

This is mostly a useful shortcut (*syntactical sugar*), but is also used by some idiomatic Python constructions.

Chapter 6

Dictionaries

This module covers the second fundamental data structure in Python: **dictionaries**, which represent a collection of *key-value pairs*. They are similar to lists, except that each element in the dictionary is also given a distinct "name" to refer to it by (instead of an index number). Dictionaries are Pythons primary version of **maps**, which is a common and extremely useful way of organizing data in a computer program—I would argue that maps are the *most useful* data structure in programming. This module will descibe how to create, access, and utilize dictionaries to organize and struture data.

Contents

6.1 Resources

- Dictionaries and Data Structures (Sweigart)
- Dictionaries (Downey)

6.2 Dictionaries

A **dictionary** is a lot like a *list*, in that it is a (one-dimensional) sequence of values that are all stored in a single variable. However, rather than using *integers* as the index for each element, a dictionary allows you to use a wide variety of different data types (including *strings* and *tuples*) as the "index". These "indices" are called **keys**, and each is used to refer to a specific **value** in the collection. Thus a dictionary is an sequence of **key-value pairs**: each element has a "key" that is used to *look up* (reference) the "value".

- This is a lot like a real-world dictionary or encyclopedia, in which the words (keys) are used to look up the definitions (values). A phone book works te same way (the names are the keys, thephone numbers are the values),
- Dictionaries provide a **mapping** of keys to values: they specify a set of data (the keys), and how that data "transforms" into another set of data (the values).

Dictionaries are written as literals inside *curly braces* (`{}`). Key-value pairs are written with a *colon* (`:`) between the key and the value, and each element (pair) in the dictionary is separated by a *comma* (`,`):

```
# a dictionary of ages
ages = {'sarah':42, 'amit':35, 'zhang':13}

# a dictionary of English words and their Spanish translation
english_to_spanish = {'one':'uno', 'two':'dos'}

# a dictionary of integers and their word representation
num_words = {1:'one', 2:'two', 3:'three'}

# like lists, dictionary values can be of different types
# including lists and other dictionaries!
type_examples = {'integer':12, 'string':'dog', 'list':[1,2,3]}

# each dictionary key can also be a different type
type_names = {'hello': 'a string', 598: 'an integer', 3.14: 'a float', (1,2): 'a tuple!'}

# dictionaries can be empty (with no elements)
empty = {}
```

- Dictionary variables are often named as plurals, but can also be named after the mapping they performed (e.g., `english_to_spanish`).
- Be careful not to name a dictionary `dict`, which is a reserved keyword (it's a function used to create dictionaries).

Dictionary *keys* can be of any hashable type (meaning the computer can consistently convert it into a number). In practice, this means that that keys are most commonly *strings*, *numbers*, or *tuples*. Dictionary *values*, on the other hand, can be of any type that you want!

Dictionary *keys* must be unique: because they are used to "look up" values, there has to be a single value associated with each key (this is called a [one-to-one mapping] in mathematics). But dictionary *values* can be duplicated: just like how two words may have the same definition in a real-world dictionary!

```
double_key = {'a': 1, 'b': 2, 'b': 3}
print(double_key)  # {'a': 1, 'b': 3}

double_val = {'a': 1, 'b': 1, 'c': 1}
print(double_val)  # {'a': 1, 'b': 1, 'c': 1}
```

Important note: dictionaries are an **unordered** collection of key-value pairs! Because you reference a value by its *key* and not by its position (as you do in a list), the exact ordering of those elements doesn't matter—the interpreter just goes immediately to the value associated with the key. This almost means that when you print out a dictionary, the order in which the elements are printed may not match the order in which you specified them in the literal (and in fact, may different between script executions or across computers!)

```
dict_a = {'a':1, 'b;':2}
dict_b = {'b':2, 'a:'1}
dict_a == dict_b  # True
```

The above examples mostly use dictionaries as "lookup tables": they provide a way of "translating" from some set of keys to some set of values. However, dictionaries are also extremely useful for grouping together related data—for example, information about a specific person:

```
person = {'first_name': "Ada", 'job': "Programmer", 'salary':78000, 'in_union':True}
```

Using a dictionary allows us to track the differnet values with named keys, rather than needing to remember whether the person's name or title was the first element!

Dictionaries can also be created from *lists* of keys and values. First use the built-in `zip()` function to create a non-list collection of tuples (each a key-value pair), and then use the built-in `dict()` function to create a dictonary out of that collection. Alternatively the built-in `enumerate()` function will create an collection with the index of each list element as its key.

```
keys = ['key0', 'key1', 'key2']
values = ['val0', 'val1', 'val2']
dict(zip(keys, values))  # {'key0': 'val0', 'key1': 'val1', 'key2': 'val2'}
dict(enumerate(values))  # {0: 'val0', 1: 'val1', 2: 'val2'}
```

6.2.1 Accessing a Dictionary

Just as with lists, we retrieve a ***value*** from a dictionary using **bracket notation**, but we put the ***key*** inside the brackets instead of the positional index (since dictionaries are unordered!).

```
# a dictionary of ages
ages = {'sarah':42, 'amit':35, 'zhang':13}

# get the value for the 'amit' key
amit_age = ages['amit']
print(amit_age)  # 35

# get the value for the 'zhang' key
zhang_age = ages['zhang']
print(zhang_age)  # 13

# accessing a key not in the dictionary will give an error
print(ages['anonymus'])  # KeyError!

# trying to look up by a VALUE will give an error (since it's not a key)
print(ages[42])  # KeyError!
```

- To reiterate: you put the *key* inside the brackets in order to access the *value*. You cannot directly put in a value in order to determine its key (because it may have more than one!)

It is worth noting that "looking up" a value by its key is a very "fast" operation (it doesn't take the interpreter a lot of time or effort). But looking up the key for a value takes time: you need to check each and every key in the dictionary to see if it has the value you're interested in!

As with lists, you can put any *expression* (including variables) inside the brackets as long as it resolves to a valid key (whether that key is a string, integer, or tuple).

As with lists, you can **mutate** (change) the dictionary by assigning values to the bracket-notation variable. This changes the *key-value pair* to have a different value, but the same key:

```
person = {'name': "Ada", 'job': "Programmer", 'salary':78000}

# assign a new value to the 'job' key
person['job'] = 'Senior Programmer'
print(person['job'])  # Senior Programmer

# assign value to itself
person['salary'] = person['salary'] * 1.15  # a 15% raise!

# add a new key-value pair by assigning a value a key
# that is not yet in the dictionary
person['shoe_size'] = 7
print(person)  # {'name': 'Ada', 'job': 'Senior Programmer', 'salary': 89700.0, 'shoe_size': 7}
```

Note that adding new elements (key-value pairs) works differently than lists: with a list, you cannot assign a value to an index that is out of bounds: you need to use the `append()` method instead). With a dictionary, you *can* assign a value to a non-existent key. This *creates* the key, assigning it the given value.

You can add a *new* key-value pair to a dictionary by assigning a value *to a key that is not yet in the dictionary*. This is distinct from lists (where you cannot assign out of bounds):

6.3 Dictionary Methods

Dictionaries support a few different operations and methods, though not as many as lists. These include:

```
# A sample dictionary to demonstrate with
sample_dict = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}

# The standard `in` operator checks for operands in the keys, not the values!
'b' in sample_dict  # True, dict contains a key `'b'`
2 in sample_dict  # False, dict does not contain a key `2`

# The get() method returns the value for the key, or a "default" value
# if the key is not in the dictionary
default = -598  # a default value
sample_dict.get('c', default)  # 3, key is in dict
sample_dict.get('f', default)  # -598, key not in dict, so return default

# Remove a key-value pair
sample_dict.pop('d')  # removes and returns the `d` key and its value

# Replace values from one dictionary with those from another
other_dict = {'a':10, 'c': 10, 'n':10}
sample_dict.update(other_dict)  # assign values from other to sample
print(sample_dict)  # {'a': 10, 'b': 2, 'c': 10, 'e': 5, 'n': 10}

sample_dict.update(a=20, c=20)  # update also supports named arguments
print(sample_dict)  # {'a': 20, 'b': 2, 'c': 20, 'e': 5, 'n': 10}

# Remove all the elements
sample_dict.clear()
```

Dictionaries also include three methods that return list-like *sequences* of the dictionary's elements:

```
sample_dict = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}

# get a "list" of the keys
sample_keys = sample_dict.keys()
print(list(sample_keys))  # ['a', 'b', 'c', 'd', 'e']

# get a "list" of the values
sample_vals = sample_dict.values()
print(list(sample_vals))  # [1, 2, 3, 4, 5]
```

```
# get a "list" of the key-value pairs
sample_items = sample_dict.items()
print(list(sample_items))  # [('a', 1), ('b', 2), ('c', 3), ('d', 4), ('e', 5)]
```

The `keys()`, `values()`, and `items()` sequences are not quite lists (they don't have all of the list operantions and methods), but they do support the `in` operator and iteration with `for` loops (see below). And as demonstrated above, they can easily be converted *into* lists if needed.

- Note that the `items()` method produces a sequence of *tuples*—each key-value pair is represented as a tuple whose first element is the key and second is the value!

6.3.1 Dictionaries and Loops

Dictionaries are iterable collections (like lists, ranges, strings, files, etc), and so you can loop through them with a **for loop**. Note that the basic `for ... in ...` syntact iterates through the dictionary's *keys* (not its values!) Thus it is much more common utilize one of the `keys()`, `values()`, or `items()` sequences.

```
sample_dict = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}

# loop through the keys (implicitly)
for key in sample_dict:
    print(key, "maps to", sample_dict[key])  # e.g., "'a' maps to 1"

# loop through the keys (explicitly)
for key in sample_dict.keys():
    print(key, "maps to", sample_dict[key])  # e.g., "'a' maps to 1"

# loop through the values. Cannot directly get the key from this
for value in sample_dict.values():
    print("someone maps to", value)  # e.g., "someone maps to 1"

# Loop through the items (each is a tuple)
for item in sample_dict.items():
    print(item[0], "maps to", item[1])  # e.g., "'a' maps to 1"
```

It is *much* more common to use **multiple assignment** to give the `items()` tuple elements local variable names, allowing you to refer to those elements by name rather than by index:

```
# Use this format instead!
for key, value in sample_dict.items():  # implicit  `key, value = item`
    print(key, "maps to", value)  # e.g., "'a' maps to 1

# Better yet, name the local variables after their semantic meaning!
for letter, number in sample_dict.items():
    print(letter, "maps to", number)  # e.g., "'a' maps to 1
```

Finally, remember that dictionaries are *unordered*. This means that there is no consistency as to which element will be processed in what order: you might get `a` then `b` then `c`, but you might get `c` then `a` then `b`! If the order is important for looping, a common strategy is to iterate through a *sorted list of the keys* (produced with the built-in `sorted()` function):

```
# Sort the keys
sorted_keys = sorted(my_dict.keys())

# Iterate through the sorted keys
for key in sorted_keys:
    print(key, "maps to", my_dict[key])

# Or all in one line!
for key in sorted(my_dict.keys()):
    print(key, "maps to", my_dict[key])
```

6.4 Nesting Dictionaries

Although dictionary *keys* are limited to hashable types (e.g., strings, numbers, tuples), dictionary *values* can be of any type—and this includes lists and other dictionaries!

Nested dictionaries are conceptually similar to nested lists, and are used in a similar manner:

```
# a dictionary representing a person (spacing is for readability)
person = {
  'first_name': 'Alice',
  'last_name': 'Smith',
  'age': 40,
  'pets': ['rover', 'fluffy', 'mittens'],  # value is an array
  'favorites': {  # value is another dictionary
    'music': 'jazz',
    'food': 'pizza',
    'numbers': [12, 42] # value is an array
  }
}

# can assign lists or dicts to a new key
person['luggage_combo'] = [1,2,3,4,5]

# person['favorite'] is an (anonymous) dictionary, so can get that dict's 'food'
favorite_food = person['favorites']['food'];

# Get to the (anonymous) 'favorites' dictionary in person, and from that get
# the (anonymous) 'numbers' list, and from that get the 0th element
first_fav_number = person['favorite']['numbers'][0];  # 12

# Since person['favorite']['numbers'] is a list, we can add to it
person['favorite']['numbers'].append(7);  # add 7 to end of the list
```

The ability to nest dictionaries inside of dictionaries is incredibly powerful, and allows us to define arbitrarily complex information structurings (schemas). Indeed, most data in computer programs—as well as public information available on the web—is structured as a set of nested maps like this (though possibly with some level of abstraction).

The other common format used with nested lists and dictionaries is to define a **list of dictionaries** where each dictionary has *the same keys* (but different values). For example:

```
# arbitrary list of people's names, heights, and weights
people = [
    {'name': 'Ada', 'height': 58, 'weight': 115},
    {'name': 'Bob', 'height': 59, 'weight': 117},
    {'name': 'Chris', 'height': 60, 'weight': 120},
    {'name': 'Diya', 'height': 61, 'weight': 123},
    {'name': 'Emma', 'height': 62, 'weight': 126}
]
```

This structure can be seen as a list of **records** (the dictionaries), each of which have a number of different **features** (the key-value pairs). This list of feature records is in fact a common way of understanding a **data table** like you would create as an Excel spreadsheet:

name	height	weight
Ada	58	115
Bob	59	117
Chris	60	120
Diya	61	123
Emma	62	126

Each dictionary (record) acts as a "row" in the table, and each key (feature) acts as a "column". As long as all of the dictionaries share the same keys, this list of dictionaries *is* a table!

- When working with large amounts of tabular data, like you might read from a `.csv` file, this is a good structure to use.

In order to analyze this kind of data table, you most often will loop through the elements in the list (the rows in the table), doing some calculations based on *each* dictionary:

```python
# How many people are taller than 60 inches?
taller_than_60 = 0
for person in people:  # iterate through the list
    # person is a dictionary
    if person['height'] >= 60:  # each dictionary has a 'height' key
        taller_than_60 += 1  # increasement count

print(taller_than_60)  # 3
```

This is effective, but not particularly efficient (or simple). We will discuss more robust and powerful ways of working with this style of data table more in a later module.

6.5 Which data structure do I use?

The last two modules have introduced multiple different **data structures** built into Python: *lists*, *tuples*, and *dictionaries*. They all represent collections of elements, though in somewhat different ways. So when do you each each type?

- Use **lists** for any *ordered* sequence of data (e.g., if you care about what comes first), or if you are collecting elements of the same general "type" (e.g., a lot of numbers, a lot of strings, etc.). If you're not sure what else to use, a *list* is a great default data structure.
- Use **tuples** when you need to ensure that a list is *immutable* (canot be changed), such as if you want it to be a key to a dictionary or a parameter to a function. Tuples are also nice if you just want to store a *small* set of data (only a few items) that is not going to change. Finally, tuples may arguable provide an "easier" syntax than lists in certain situations, such as when using multiple assignment.
- Use **dictionaries** whenever you need to represent a *mapping* of data and want to link some set of keys to some set of values. If you want to be able to "name" each value in your collection, you can use a dictionary. If you want to work with key-value pairs, you need to use a dictionary (or some other dictionary-like data structure)

Chapter 7

Functional Iteration

This module introduces techniques from **Functional Programming**, which is a programming paradigm centered on *functions* rather than on *variables* and statements as we've been doing so far (known as *imperative programming*). Functional programming offers another way to think about giving "instructions" to a computer, which can make it easier to think about and implement some algorithms. While not completely functional language, Python does contain a number of "functional-programming-like" features that can be mixed with the imperative strategies we're used to, allowing for more compact and readable code in some cases.

Contents

7.1 Resources

- Functional Programming in Python (IBM) (note: Python 2)
- Map, Filter, Lambda, and List Comprehensions in Python (note: Python 2)
- Functional Programming in Python (O'Reilly) (short eBook)
- List Comprehensions (Python Docs)
- List Comprehensions Explained Visually
- Functional Programming HOWTO (advanced, not recommended)

7.2 Functions ARE Variables

Previously we've described functions as "named sequences of instructions", or groupings of lines of code that are given a name. But in a functional programming paradigm, functions are *first-class objects*—that is, they are "things" (values) that can be organized and manipulated *just like variables*.

In Python, **functions ARE variables**:

```python
# create a function called `say_hello`
def say_hello(name):
    print("Hello, "+name)

# what kind of thing is `say_hello` ?
type(say_hello)  # <class 'function'>
```

Just like `x = 3` defines a variable for a value of type `int`, or `msg = "hello"` defines a variable for a value of type `string`, the above `say_hello` function is actualy a variable for a *value* of type `function`!

- This is why it is accidentally possible to "overwrite" built-in functions by assigning values to variables like `sum`, `max`, or `dict`.
- Note that we refer to the function by its name *without* the parentheses!

The fact that functions **are** variables is the core realization to make when programming in a functional style. You need to be able to think about functions as **things** (nouns), rather than as **behaviors** (objects). If you imagine that functions are "recipes", then you need to think about them as *pages from the cookbook* (that can be bound together or handed to a friend), rather than just the sequence of actions that they tell you to perform.

And because functions are just another type of variable, they can be used **anywhere** that a "regular" variable can be used. For example, functions are values, so they can be assigned to other variables!

```python
# create a function `say_hello`
def say_hello(name):
    print("Hello, "+name)

# assign the `say_hello` value to a new variable `greet`
greet = say_hello

# call the function assigned to the `greet` variable
greet("world")  # prints "Hello world"
```

- It helps to think of functions as just a special kind of list. Just as *lists* have a special syntax `[]` (bracket notation) that can be used to "get" a value from the list, *functions* have a special syntax `()` (parentheses) that can be used to "run" the function.

Moreover, functions are values, so they can be *passed as parameters to other functions*!

```python
# create a function `say_hello`
def say_hello(name):
    print("Hello, "+name)

# a function that takes ANOTHER FUNCTION as an argument
# this function will call the argument function, passing it "world"
def do_with_world(func_to_call):
  # call the given function with an argument of "world"
  func_to_call("world")

# call `do_with_world`, saying the "thing to do" is `say_hello`
do_with_world(say_hello)  # prints "Hello world"
```

In this case, the `do_with_world` function will *execute* whatever function it is given, passing in a value of `"world"`. (You can think of this as similar to having a function that accesses the `'world'` key of a given dictionary).

- **Important note**: when we pass `say_hello` as an argument, we don't put any parentheses after it! Putting the parentheses after the function name *executes* the function, causing it to perform the lines of code it defines. This will cause the expression containing the function to *resolve* to its returned value, rather than being the function value itself. It's like passing in the baked cake rather than the recipe page.

"'python def greet(): # version with no args return "Hello"

# print out the function print(say_hello) # prints , the function

# resolve the expression, then print that out print(say_hello()) # prints "Hello", which is what `say_hello()` resolves to. "'

A function that is passed into another is commonly referred to as a **callback function**: it is an argument that the other function will "call back to" and execute when needed.

Functions can take more than one *callback function* as arguments, which can be a useful way of *composing* behaviors.

```python
def do_at_once(first_callback, second_callback):
    first_callback()  # execute the first function
    print("and", end=" ")
    second_callback()  # execute the second function
    print("at the same time! ")

def pat_head():
    print("pat your head", end=" ")

def rub_belly():
    print("rub your belly", end=" ")

# pass in the callbacks to "do at once"
do_at_once(pat_head, rub_belly)
```

This idea of *passing functions are arguments to other functions* is at the heart of functional programming, and is what gives it expressive power: we can define program behavior primarily in terms of the behaviors that are run, and less in terms of the data variables used.

7.2.1 lambdas: Anonymous Functions

We have previously used **anonymous variables** in our programs, or values which are not assigned a variable name (so remain anonymous). These values were defined as *literals* or expressions and passed directly into functions, rather than assigning them to variables:

```python
my_list = [1,2,3]  # a named variable (not anonymous)
print(my_list)  # pass in non-anonymous variable
print([1,2,3])  # pass in anonymous value
```

Because functions **are** variables, it is also possible to define **anonymous functions**: functions that are not given a name, but instead are passed directly into other functions. In Python, these anonymous functions are referred to as **lambdas** (named after lambda calculus, which is a way of defining algorithms in terms of functions).

Lambdas are written using the following general syntax:

```python
lambda arg1, arg2: expression_to_return
```

We indicate that we are defining a lambda function with the keyword `lambda` (rather than the keyword `def` used for named functions). This is followed by a list of arguments separated by commas (what normally goes inside the `()` parenthes in a named function definition), then a colon `:`, then the expression that will be *returned* by the anonymous function.

For example, compare the following named and anonymous function definitions:

```
# named function to square a value
def square(x):
    return x**2

# anonymous function to square a value (assigned to a variable)
lambda x: x**2

# named function to combine first and last name
def make_full_name(first, last):
    return first + " " + last

# anonymous function to combine first and last name
lambda first, last: first + " " + last
```

- We're basically replacing `def` and the function name with the word `lambda`, removing the parentheses around the arguments, and removing the `return` keyword!

Just as other expressions can be assigned to variables lambda functions can be assigned to variables in order to give them a name. This is the equivalent of having defined them as named functions in the first place:

```
square = lambda x: x**2
make_full_name = lambda first, last: first + " " + last
```

There is one major restriction on what kind of functions can be defined as anonymous lambdas: they must be functions that consist of **only** a single returned expression. That is, they need to be a function that contains exactly one line of code, which is a `return` statement (as in the above examples). This means that lambdas are ***short*** functions that usually perform very simple transformations to the arguments... exactly what we want to do with functional programming!

7.3 Functional Looping

Why do we care about treating functions as variables, or defining anonymous lambda functions? Because doing so allows us to *replace loops with function calls* in some situations. For particular kinds of loops, this can make the code more *expressive* (more clearly indicative of what it is doing).

7.3.1 Map

For example, consider the following loop:

```
def square(n): # a function that squares a number
    return n**2

numbers = [1,2,3,4,5]  # an initial list

squares = []  # the transformed list
for number in numbers:
    transformed = square(number)
    squares.append(transformed)
print(squares)  # [1, 4, 9, 16, 25]
```

This loop represents a **mapping** operation: it takes an original list (e.g., of numbers 1 to 5) and produces a *new* list with each of the original elements transformed in a certain way (e.g., squared). This is a common operation to apply: maybe you want to "transform" a list so that all the values are rounded or lowercase, or you want to *map* a list of words to a list of their lengths. It is possible to make these changes uses the same pattern as above: create an empty list, then loop through the original list and `append` the transformed values to the new list.

However, Python also provides a *built-in function* called `map()` that directly peform this kind of mapping operation on a list without needing to use a loop:

```python
def square(n):  # a function that squares a number
    return n**2

numbers = [1,2,3,4,5]  # an initial list

squares = list(map(square, numbers))
print(squares)  # [1, 4, 9, 16, 25]
```

The `map()` function takes a list a produces a *new* list with each of the elements transformed. The `map()` function takes in two arguments: the second is the list to transform, and the first is the *name of a callback function* that will do the transformation. This callback function must take in a *single* argument (an element to transform) and return a value (the transformed element).

- Note that in Python 3, the `map()` function returns an *iterator*, which is a list-like sequence similar to that returned by a dictionary's `keys()` or `items()` methods. Thus in order to interact with it as a list, it needs to be converted using the `list()` function.

The `map()` callback function (e.g., `square()` in the above example) can also be specified using an anonymous lambda, which allows for concisely written code (but often at the expense of readability—see *List Comprehensions* below for a more elegant, Pythonic solution).

```python
numbers = [1,2,3,4,5]  # an initial list
squares = list(map(lambda n:n**2, numbers))
```

7.3.2 Filter

A second common operation is to **filter** a list of elements, removing elements that we don't want (or more accurately: only keeping elements that we DO want). For example, consider the following loop:

```python
def is_even(n):  # a function that determines if a number is even
    remainder = n % 2  # get remainder when dividing by 2 (modulo operator)
    return remainder == 0  # True if no remainder, False otherwise

numbers = [2,7,1,8,3]  # an initial list

evens = []  # the filtered list
for number in numbers:
    if is_even(number):
      evens.append(number)
print(evens)  # [2, 8]
```

With this **filtering** loop, we are *keeping* the values for which the `is_even()` function returns true (the function determines "what to let in" not "what to keep out"), which we do by appending the "good" values to a new list.

Similar to `map()`, Python provides a *built-in function* called `filter()` that will directly perform this filtering:

```python
def is_even(n):  # a function that determines if a number is even
    return (n % 2) == 0  # True if no remainder, False otherwise

numbers = [2,7,1,8,3]  # an initial list
```

```
evens = list(filter(is_even, numbers))
print(evens)  # [2, 8]
```

The `filter()` function takes a list a produces a *new* list that contains only the elements that *do match* a specific criteria. The `filter()` function takes in two arguments: the second is the list to filter, and the first is the *name of a callback function* that will do the filtering. This callback function must take in a *single* argument (an element to consider) and return `True` if the element should be included in the filtered list (or `False` if it should not be included).

Because `map()` and `filter()` both produce list-like sequences, it is possible to take the returned value from one function and pass it in as the argument to the next. For example:

```
numbers = [1,2,3,4,5,6]

# get the squares of EVEN numbers only
filtered = filter(is_even, numbers)  # filter the numbers
squares = map(square, filtered)  # map the filtered values
print(list(squares))  # [4, 16, 36]

# or in one statement, passing results anonymously
squares = map(square,
          filter(is_even,
          numbers))  # watch out for the parentheses!
print(list(squares))  # [4, 16, 36]
```

This structure can potentially make it easier to understand the code's intent: it is "`square`ing the `is_even` `numbers`"!

7.3.3 Reduce

The third important operation in functional programming (besides *mapping* and *filtering*) is **reducing** a list. Reducing a list means to *aggregate* that lists values togther, transforming the list into a single value. For example, the built-in `sum()` function is a *reducing* operation (and in fact, the most common one!): it reduces a list of numbers to a single summed value.

- You can think of `reduce()` as a *generalization* of the `sum()` function—but rather than just adding (`+`) the values together, `reduce()` allows you to specify what operation to perform when aggregating (e.g., multiplication).

Because the `reduce()` function can be complex to interpret, it was actually *removed* from the set of "core" built-in functions in Python 3 and relegated to the `functools` module. Thus we need to import the function in order to use it:

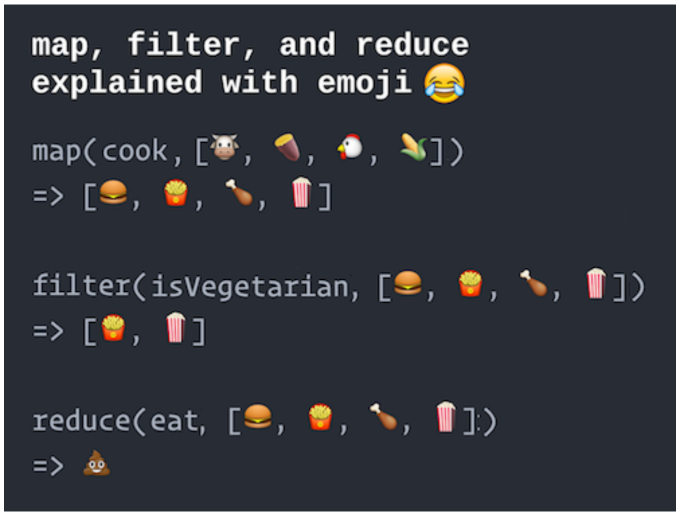
```
from functools import reduce
```

To understand how a *reduce* operation works, consider the following basic loop:

```
def multiply(x, y): # a function that multiplies two numbers
    return x*y

numbers = [1,2,3,4,5]  # an initial list

running_total = 1  # an accumulated aggregate
for number in numbers:
    running_total = multiply(running_total, number)
print(running_total)  # 120  (1*2*3*4*5)
```



```
map, filter, and reduce
explained with emoji 😂

map(cook, [🐮, 🍠, 🐔, 🌽])
=> [🍔, 🍟, 🍗, 🍿]

filter(isVegetarian, [🍔, 🍟, 🍗, 🍿])
=> [🍟, 🍿]

reduce(eat, [🍔, 🍟, 🍗, 🍿])
=> 💩
```

Figure 7.1: Map, filter, reduce explained with emoji

This loop **reduces** the list into an “accumulated” product (factorial) of all the numbers in the list. Inside the loop, the `multiply()` function is called and passed the “current total” and the “new value” to be combined into the aggregate (*in that order*). The resulting total is then reassigned as the “current total” for the next iteration.

The `reduce()` function does exactly this work: it takes as arguments a *callback* function used to combine the current running total with the new value, and a list of values to combine. Whereas the `map()` and `filter()` callback functions each took 1 argument, the `reduce()` callback function requires **2** arguments: the first will be the “running total”, and the second will be the “new value” to mix into the aggregate. (While this ordering doesn’t influence the factorial example, it is relevant for other operations):

```python
def multiply(x, y): # a function that multiplies two numbers
    return x*y

numbers = [1,2,3,4,5]  # an initial list

product = reduce(multiply, numbers)
print(product)  # 120
```

- The `reduce()` function aggregates into a single value, so the result doesn’t need to be converted from an *iterator* to a list!

To summarize, the `map()`, `filter()`, and `reduce()` operations work as follows:

All together, the **map**, **filter**, and **reduce** operations form the basic for a functional consideration of a program. Indeed, these kinds of operations are very common when discussing data manipulations: for example, the famous MapReduce model involves “mapping” each element through a complex function (on a different computer no less!), and then “reducing” the results into a single answer.

7.4 List Comprehensions

While `map()` and `filter()` are effective ways of producing new lists from old, they can be somewhat hard to read (particularly when using anonymous lambda functions, which we often would want to do for simple transformations). Instead, a more idomatic and “Pythonic” approach (preferred by language developer Guideo van Rossum) is to use **List Comprehensions**. A *list comprehesion* is a special syntax for doing mapping and/or filtering operations on list using the `for` and `if` keywords you are familiar with.

A basic list comprehension has the following syntax:

```
new_list = [output_expression for loop_variable in sequence]
```

For example, a list comprehsion to **map** from a list of numbers to their squares would be:

```
numbers = [1,2,3,4,5]  # original list
squares = [n**2 for n in numbers]
print(squares)  # [1, 4, 9, 16, 25]
```

List comprehensions are written inside square brackets `[]` and use the same `for ... in ...` syntax used in for loops. However, the *expession* that you would normally `append()` to the output list when mapping (or that is returned from an anonymous lambda function) is written *before* the `for`. This causes the above comprehension to be read as *"a list consisting of `n**2` (n squares) for each `n` in `numbers`"*—it's almost English!

You can contrast a list comprehension with the same mapping operation done via a loop or via a `map()` and a lambda:

```
# with a loop
squares = []
for n in numbers:
    squares.append(n**2)  # append expression

# with a lambda
squares = list(map(lambda n: n**2, numbers))  # map with lambda

# with a list comprehesion
squares = [n**2 for n in numbers]  # map with list comprehension
```

Notice that all 3 versions specify a *transformation expression* (`n**2`) on a input variable (`n`). They just use different syntax (punctuation and ordering) to specify the transformation that should occur.

List comprehensions can also be used to **filter** values (even as they are being mapped). This is done by specifying an `if` filtering condition after the sequence:

```
new_list = [output_expression for loop_variable in sequence if condition]
```

Or as a specific example (remember: we filter for elements to *keep*!):

```
numbers = [2,7,1,8,3]
evens = [n for n in numbers if n%2 == 0]
print(evens)  # [2, 8]
```

This can be read as *"a list consisting of `n` for each `n` in `numbers`, but only `if n%2 == 0`"*. It is equivalent to using the `for` loop:

```
evens = []
for n in numbers
    if n%2 == 0:  # check the filter condition
        evens.append(n)  # append the expression
```

Finally, it is possible to include *multiple, nested* `for` and `if` statements in a list comprehension. Each successive `for ... in ...` or `if` expression is included inside the square brackets after the output expression: This allows you to effectively convert nested control structures into a comprehension:

```
entrees = ["chicken","fish","veggies"]
sides = ["potatoes", "veggies"]

# get all "meals" if the entree and side are not the same
```

```
meals = [ entree+" & "+side for entree in entrees for side in sides if entree != side]
print(meals)  # ['chicken & potatoes', 'chicken & veggies', 'fish & potatoes',
              #  'fish & veggies', 'veggies & potatoes']
              # note: no "veggies and veggies" !
```

This is equivalent to the nested loops:

```
meals = []
for entree in entrees:
    for side in sides:
        if entree != side:
            meals.append(entree+" & "+side)
```

(This *almost* acts like a **reduce** operation, reducing two lists into a single one... but it doesn't exactly convert).

Overall, list comprehensions are considered a *better, more Pythonic* approach to functional programming. However, `map()` `filter()` and `reduce()` functions are a more generalized approach that can be found in multiple different languages and contexts, including other data-processing languages such as R, Julia, and JavaScript. Thus it is good to be at least familiar with both approaches!

Chapter 8

Pandas

This module introduces the *Python Data Analysis* library `pandas`—a set of modules, functions, and classes used to for easily and efficiently performing data analysis—`panda`'s speciality is its highly optimized performance when working with large data sets. `pandas` is the most common library used with Python for Data Science (and mirrors the `R` language in many ways, allowing programmers to easily move between the two). In this module, we will discuss the two main data structures used by `pandas` (*Series* and *DataFrames*) and how to use them to organize and work with data.

Contents

8.1 Resources

- 10 minutes to pandas (pandas docs) a basic set of examples
- Tutorials (pandas docs) a list and guide to various tutorials (of mixed quality)
- Intro to Data Structure (pandas docs)
- Essential Basic Functionality (pandas docs) not really basic, but a complete set of examples
- Pandas. Data Processing (Data Analysis in Python)
- pandas Foundations (DataCamp)

8.2 Setup

`pandas` is a **third-party** library (not built into Python!), but is included by default with Anaconda and so can be imported directly. Additionally, Pandas is built on top of the `numpy` scientific computing library which supports highly optimized mathematical operations. Thus many `pandas` operations involve working with `numpy` data structures, and the `pandas` library requires `numpy` (also included in Anaconda) to also be imported:

```
# import libraries
import pandas as pd  # standard shortcut names
import numpy as np
```

- We usually `import` the module and reference types and methods using dot notation, rather than importing them into the global namespace.
- Note that this module will focus primarily on `pandas`, leaving `numpy`-specific data structures and functions for the reader to explore.

8.3 Series

The first basic `pandas` data structure is a **Series**. A Series represents a *one-dimensional ordered collection of values*, making them somewhat similar to a regular Python *list*. However, elements can also be given *labels* (called the **index**), which can be non-numeric values, similar to a *key* in a Python *dictionary*. This makes a Series somewhat like an ordered dictionary—one that supports additional methods and efficient data-processing behaviors.

Series can be created using the `Series()` function (a *constructor* for instances of the class):

```
# create a Series from a list
number_series = pd.Series([1, 2, 2, 3, 5, 8])
print(number_series)
```

produces

```
0    1
1    2
2    2
3    3
4    5
5    8
dtype: int64
```

Printing a Series will display it like a *table*: the first value in each row is the **index** (label) of that element, and the second is the value of the element in the Series.

- Printing will also display the *type* of the elements in the Series. All elements in the Series will be treated as "same" type—if you create a Series from mixed elements (e.g., numbers and strings), the type will be the a generic `object`. In practice, we almost always create Series from a single type.

If we create a Series from a list, each element will be given an *index* (label) that is that values's index in the list. We can also create a Series from a *dictionary*, in which case the keys will be used as the index labels:

```
# create a Series from a dictionary
age_series = pd.Series({'sarah':42, 'amit':35, 'zhang':13})
print(age_series)
```

```
amit     35
sarah    42
zhang    13
dtype: int64
```

- Note that the Series is automatically **sorted** by the keys of the dictionary! This means that the order of the elements in the Series will always be the same for a given dictionary (which cannot be said for the dictionary items themselves).

8.3.1 Series Operations and Methods

The main benefit of Series (as opposed to normal lists or dictionaries) is that they provide a number of operations and methods that make it easy to consider and modify the entire Series, rather than needing to worth with each element individually. In a way, the functions include built-in *mapping*, *reducing*, and *filtering* style operations.

In particular, basic operators (whether math operators such as `+` and `-`, or relational operators such as `>` or `==`) function as **vectorized operations**, meaning that they are applied to the entire Series **member-wise**: the operation is applied to the first element in the Series, then the second, then the third, and so forth:

```
sample = pd.Series(range(1,6))  # Series of numbers from 1 to 5 (6 is excluded)
result = sample + 4  # add 4 to each element (produces new Series)
print(result)
    # 0    5
    # 1    6
    # 2    7
    # 3    8
    # 4    9
    # dtype: int64

is_greater_than_3 = sample > 3  # compare each element
print(is_greater_than_3)
    # 0    False
    # 1    False
    # 2    False
    # 3     True  # note index and value are not the same
    # 4     True
    # dtype: bool
```

- Having a Series operation apply to a *scalar* (a single value) is referred to as **broadcasting**. The idea is that the smaller "set" of elements (e.g., a single value) is *broadcast* so that it has a comparible size, thereby allowing different "sized" data structures to interact. Technically, operating on a Series with a *scalar* is actually a specific case of operating on it with another Series!

If the second operand is *another Series*, then mathematical and relational operations are still applied **member-wise**, with the elements of each operand being "matched" by their index label. This means that for most Series whose indices are

```
s1 = pd.Series([2, 2, 2, 2, 2])
s2 = pd.Series([1, 2, 3, 4, 5])

# examples of operations (list only includes values)
list(s1 + s2)  # [3, 4, 5, 6, 7]
list(s1 / s2)  # [2.0, 1.0, 0.66666666666666663, 0.5, 0.40000000000000002]
list(s1 < s2)  # [False, False, True, True, True]

# add a Series to itself (why not?)
list(s2 + s2)  # [2, 4, 6, 8, 10]

# perform more advanced arithmetic!
s3 = (s1 + s2) / (s1 + s1)
list(s3)  # [0.75, 1.0, 1.25, 1.5, 1.75]
```

And note that these operations will be *fast*, even for very large Series, allowing for effective data manipulations.

`pandas` Series also include a number of *methods* for inspecting and manipulating the data. Some useful examples (not comprehensive):

Function	Description
`index`	an *attribute*; the sequence of index labels (convert to a *list* to use)
`head(n)`	returns a Series containing only the first `n` elements
`tail(n)`	returns a Series containing only the last `n` elements
`any()`	returns whether ANY of the elements are `True` (or "truthy")

Function	Description
`all()`	returns whether ALL of the elements are `True` (or “truthy”)
`mean()`	returns the statistical mean of the elements in the Series
`std()`	returns the standard deviation of the elements in the Series
`describe()`	returns a Series of descriptive statistics

Function	Description
`idxmax()`	returns the index label of the element with the max value

Series support many more methods as well: see the full documentation for a complete list.

One particularly useful method to mention is the `apply()` method. This method is used to *apply* a particular **callback function** to each element in the series. This is a *mapping* operation, similar to what we've done with the `map()` function:

```
def square(n):  # a function that squares a number
    return n**2

number_series = pd.Series([1,2,3,4,5])  # an initial series

square_series = number_series.apply(square)
list(square_series)  # [1, 4, 9, 16, 25]

# can also apply built-in functions
import math
sqrt_series = number_series.apply(math.sqrt)
list(sqrt_series)  # [1.0, 1.4142135623730951, 1.7320508075688772, 2.0, 2.23606797749979]

# pass additional arguments as keyword args (or `args` for a single argument)
cubed_series = number_series.apply(math.pow, args=(3,)) # call math.exp(n, 3) on each
list(cubed_series)  # [1.0, 8.0, 27.0, 64.0, 125.0]
```

8.3.2 Accessing Series

Just like lists and dictionaries, elements in a Series can be accessed using **bracket notation**, putting the index label inside the brackets:

```
number_series = pd.Series([1, 2, 2, 3, 5, 8])
age_series = pd.Series({'sarah':42, 'amit':35, 'zhang':13})

# get the 1th element from the number_series
number_series[1]  # 2

# get the 'sarah' element from age_series
age_series['amit']  # 35

# get the 0th element from age_series
# (Series are ordered, so can be accessed positionally)
age_series[0]  # 42
```

Note that the returned values are not technically basic `int` or `float` or `string` types, but are rather specific `numpy` objects that work almost identically to their normal type (but with some additional optimization).

You can also use list-style *slices* using the colon operator (e.g., elements `1:3`).

it is also possible to specify ***a sequence of indicies*** (i.e., a *list* or *range* or even a *Series* of indices) to access using bracket notation. This will produce a new Series object that contains only the elements that have those labels:

```
age_series = pd.Series({'sarah':42, 'amit':35, 'zhang':13})

index_list = ['sarah', 'zhang']
print( age_series[index_list] )
    # sarah    42
    # zhang    13
    # dtype: int64

# using an anonymous variable for the index list (notice the brackets!)
print( age_series[['sarah', 'zhang']] )
    # sarah    42
    # zhang    13
    # dtype: int64
```

This also means that you can use something like a *list comprehension* to (or even a Series operation!) to determine which elements to select from a Series!

```
letter_series = pd.Series(['a','b','c','d','e','f'])
even_numbers = [num for num in range(0,6) if num%2 == 0]  # list of even numbers

# get letters with even numbered indices
letter_series[even_numbers]  # []
    # 0    a
    # 2    c
    # 4    e
    # dtype: object

# in one line (check the brackets!)
letter_series[[num for num in range(0,6) if num%2 == 0]]
```

Finally, using a ***sequence of booleans*** with bracket notatoin will produce a new Series containing the elements whose position *corresponds* with `True` values. This is called **boolean indexing**.

```
shoe_sizes = pd.Series([7, 6.5, 4, 11, 8])  # a series of shoe sizes
index_filter = [True, False, False, True, True]  # list of which elements to extract

# extract every element in an index that is True
shoe_sizes[index_filter]  # has values 7.0, 11.0, 8.0
```

- In this example, since `index_filter` is `True` at index 0, 3, and 4, then `shoe_sizes[index_filter]` returns a Series with the elements from index numbers 0, 3, and 4.

This is incredibly powerful because it allows us to easily perform **filtering** operations on a Series:

```
shoe_sizes = pd.Series([7, 6.5, 4, 11, 8])  # a series of shoe sizes
big_sizes = shoe_sizes > 6.5  # has values True, False, False, True, True
```

```
big_shoes = shoe_sizes[big_sizes]  # has values 7, 11, 8

# as one line
big_shoes = shoe_sizes[shoe_size > 6.5]
```

- You can think of the last statement as saying *shoe sizes* ***where*** *shoe size is greater than 6.5*.
- You can include *logical operators* ("and" and "or") by using the operators `&` for "and" and `|` for "or". Be sure to wrap each relational expression in `()` to enforce order of operations.

While it is perfectly possible to do similar filtering with a list comprehension, the boolean indexing expression can be very simple to read and runs quickly. (This is also the normal style of doing filtering in the `R` programming language).

8.4 Data Frames

The most common data structure used in `pandas` (more common than Series) is a **DataFrame**. A DataFrame represents a **table**, where data is organized into rows and columns. You can think of a DataFrame as being like a Excel spreadsheet or a SQL table.

- We have previously represented tabular data using a *list of dictionaries*. However, this required us to be careful to make sure that all of the dictionaries shared keys, and did not offer easy ways to interact with the table in terms of its rows or columns. DataFrames give us that functionality!

A DataFrame can also be understood as a *dictionary of Series*, where each Series represents a **column** of the table. The keys of this dictionary are the *index labels* of the columns, while the the index labels of the Series serve as the labels for the row.

- This is distinct from spreadsheets or SQL tables, which are often seen as a collection of *observations* (rows). Programmatically, DataFrames should primarily be considered as a collection of *features* (columns), which happen to be sequenced to correspond to observations.

A DataFrame can be created using the `DataFrame()` function (a *constructor* for instances of the class). This function usually takes as an argument *dictionary* where the values are Series (or values that can be converted into a Series, such as a list or a dictionary):

```
name_series = pd.Series(['Ada','Bob','Chris','Diya','Emma'])
heights = range(58,63)
weights = [115, 117, 120, 123, 126]

df = pd.DataFrame({'name':name_series, 'height':heights, 'weight':weights})
print(df)
    #    height   name  weight
    # 0      58    Ada     115
    # 1      59    Bob     117
    # 2      60  Chris     120
    # 3      61   Diya     123
    # 4      62   Emma     126
```

- Although DataFrames variables are often named `df` in `pandas` examples, this is ***not*** a good variable name. You should use much more descriptive names for your DataFrames (e.g., `person_size_table`) when used in actual programs.
- You can specify the order of columns in the table using the `columns` keyword argument, and the order of the rows using the `index` keyword argument.

It is also possible to create a DataFrame directly from a spreadsheet—such as from `.csv` file (containing **c**omma **s**seperated **v**alues) by using the `pandas.read_csv()` function:

```
my_dataframe = pd.read_csv('path/to/my/file.csv')
```

See the IO Tools documentation for details and other file-reading functions.

8.4.1 DataFrame Operations and Methods

Much like Series, DataFrames support a **vectorized** form of mathematical and relational operators: when the other operand is a *scalar*, then the operation is applied member-wise to each value in the DataFrame:

```
# data frame of test scores
test_scores = pd.DataFrame({
    'math':[91, 82, 93, 100, 78, 91],
    'spanish':[88, 79, 77, 99, 88, 93]
})

curved_scores = test_scores * 1.02  # curve scores up by 2%
print(curved_scores)
    #      math  spanish
    # 0   92.82    89.76
    # 1   83.64    80.58
    # 2   94.86    78.54
    # 3  102.00   100.98
    # 4   79.56    89.76
    # 5   92.82    94.86

print(curved_scores > 90)
    #     math spanish
    # 0   True   False
    # 1  False   False
    # 2   True   False
    # 3   True    True
    # 4  False   False
    # 5   True    True
```

It is possible to have both operands be DataFrames. In thiis case the operation is applied member-wise, where values are matched if they have the same row and column label. Note that any value that doesn't have a pair will instead produce the value `NaN` (Not a Number). This is not a normal way of working with DataFrames—it is much more common to access individual rows and columns and work with those (e.g., add make a new column that is the sum of two others); see below for details.

Also like Series, DataFrames objects support a large number of methods, including:

Function	Description
`index`	an *at-tribute*; the se-quence of **row** index la-bels (con-vert to a *list* to use)
`columns`	an *at-tribute*; the se-quence of **col-umn** index la-bels (con-vert to a *list* to use)
`head(n)`	returns a DataFrame con-tain-ing only the first `n` *rows*

Function	Description
`tail(n)`	returns a DataFrame containing only the last `n` *rows*
`assign(...)`	returns a new DataFrame with an additional column; call as `df.assign(new_label=new_column)`
`drop(label, row_or_col)`	returns a new DataFrame with the given row or column removed

Function	Description
`mean()`	returns a Series of the statistical means of the values of each **column**
`all()`	returns a Series of whether ALL the elements in each **column** are `True` (or “truthy”)

Function	Description
`describe()`	returns a DataFrame whose columns are Series of descriptive statistics for each **column** in the original DataFrame

You may notice that many of these methods (e.g., `head()`, `mean()`, `describe()`, `any()`) also exist for Series. In fact, most every method that Series support are supported by DataFrames as well. These methods are all applied **per column** (not per row)—that is, calling `mean()` on a DataFrame will calculate the *mean* of **each column** in that DataFrame:

```
df = pd.DataFrame({
    'name':['Ada','Bob','Chris','Diya','Emma'],
    'height':range(58,63),
    'weights':[115, 117, 120, 123, 126]})
df.mean()
    # height      60.0
    # weights    120.2
    # dtype: float64
```

If the Series method would return a *scalar* (a single value, as with `mean()` or `any()`), then the DataFrame method returns a Series whose labels are the column labels, as above. If the Series method instead would return a Series (multiple values, as with `head()` or `describe()`), then the DataFrame method returns a new DataFrame whose columns are each of the resulting Series:

```
df = pd.DataFrame({
    'name':['Ada','Bob','Chris','Diya','Emma'],
    'height':range(58,63),
    'weights':[115, 117, 120, 123, 126]})
df.describe()
    #         height     weights
    # count   5.000000    5.000000
    # mean   60.000000  120.200000
    # std     1.581139    4.438468
    # min    58.000000  115.000000
    # 25%    59.000000  117.000000
    # 50%     60.000000  120.000000
    # 75%     61.000000  123.000000
    # max     62.000000  126.000000
```

- Notice that the `height` column is the result of calling `describe()` on the DataFrame's `height` column Series!
- As a general rule: if you're expecting one value per column, you'll get a Series of those values; if you're expecting multiple values per column, you'll get a DataFrame of those values.
- This also means that you can sometimes "double-call" methods to reduce them further. For example, `df.any()` returns a Series of whether each column contains a `True` value; `df.all().all()` would check if *that* Series contains all `True` values (thus checking *all* columns have all `True` value, i.e., the entire table is all `True` values).

8.4.2 Accessing DataFrames

DataFrames make it possible to quickly access individual or a subset of values, though these methods use a variety of syntax structures. For this explanation, refer to the following sample DataFrame initially described above:

```
# all examples in this section
df = pd.DataFrame({
    'name':['Ada','Bob','Chris','Diya','Emma'],
    'height':range(58,63),
    'weight':[115, 117, 120, 123, 126]
})

print(df)
    #      height   name   weight
    # 0       58     Ada      115
    # 1       59     Bob      117
    # 2       60   Chris      120
    # 3       61    Diya      123
    # 4       62    Emma      126
```

Since DataFrames are most commonly viewed as a *dictionary of columns*, it is possible to access them as such using **bracket notation** (using the index label of the column):

```
print( df['height'] )  # get height column
    # 0    58
    # 1    59
    # 2    60
    # 3    61
    # 4    62
    # Name: height, dtype: int64
```

However, it is often more common to refer to individual columns using **dot notation**, treating each column as an *attribute* or *property* of the DataFrame object:

```
# same results as above
print( df.height )  # get height column
```

It is also possible to select *multiple* columns by using a *list* or sequence inside the **bracket notation** (similar to selecting multiple values from a Series). This will produce a new DataFrame (a "sub-table")

```
# count the brackets carefully!
print( df[['name', 'height']] )  # get name and height columns

# can also select multiple columns with a list of their positions
print( df[[1,2]] )  # get 1st (name) and 2nd (weight) columns
```

- *Watch out though*! Specifying a **slice** (using a colon `:`) will actually select by *row* position, not column position!

```
python   print( df[0:2] ) # get ROWS 0 through 2 (not inclusive)       #    height name
weight       # 0      58  Ada      115       # 1      59  Bob      117
```

I do not know wherefore this inconsistency, other than “convenience”.

Because DataFrames support multiple indexes, it is possible to use **boolean indexing** (as with Series), allowing you to *filter* for rows based the values in their columns:

```
print( df[ df.height > 60 ] )
    #    height  name  weight
    # 3      61  Diya     123
    # 4      62  Emma     126
```

- Note that `df.height`is a Series (a column), so `df.height > 60` produces a Series of boolean values (`True` and `False`). This Series is used to determine *which* rows to return from the DataFrame—each row that corresponds with a `True` index.

Finally, DataFrames also provide two *attributes* (properties) used to “quick access” values: **`loc`**, which provides an “index” (lookup table) based on index labels, and **`iloc`**, which provides an “index” (lookup table) based on row and column positions. Each of these “indexes” can be thought of as a *dictionary* whose values are the individual elements in the DataFrame, and whose keys can therefore be used to access those values using **bracket notation**. The dictionaries support multiple types of keys (using label-based `loc` as an example):

Key Type	Description	Example
`df.loc[row_label]`	An individual value	`df.loc['Ada']` (the row labeled `Ada`)
`df.loc[row_label_list]`	A list of row labels	`df.loc[['Ada','Bob']]` (the rows labeled `Ada` and `Bob`)

Key Type	Description	Example
`df.loc[row_label_slice]`	A *slice* of row labels	`df.loc['Bob':'Diya']` (the rows from `Bob` to `Diya`. Note that this is an *inclusive* slice!)
`df.loc[row_label, col_label]`	A *tuple* of `(row, column)`	`df.loc['Ada', 'height']` (the value at row `Ada`, column `height`)
`df.loc[row_label_seq, col_label_seq]`	A *tuple* of label lists or slices	`df.loc['Bob':'Diya', ['height','weight']]` (the rows from `Bob` to `Diya` with the columns `height` and `weight`)

- Note that the example `df` table doesn't have row labels beyond `0` to `4`
- Using a *tuple* makes it easy to access a particular value in the table, or a range of values (*selecting* rows and columns).
- Note that we can also use the boundless slice `:` to refer to "all elements". So for example:

`python    df.loc[:, 'height']  # get all rows, 'height' column`

This is a basic summary of how to create and access DataFrames; for more detailed usage, additional methods, and specific "recipes", see the official `pandas` documentation.

Bibliography