# 1.2 A First Program

This section discusses the creation of your first program, running the interpreter, and some basic debugging.

### Running Python

Python programs always run inside an interpreter.

The interpreter is a "console-based" application that normally runs from a command shell.

python3

Python 3.6.1 (v3.6.1:69c0db5050, Mar 21 2017, 01:21:04)

[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin

Type "help", "copyright", "credits" or "license" for more information.

>>>

Expert programmers usually have no problem using the interpreter in this way, but it's not so user-friendly for beginners. You may be using an environment that provides a different interface to Python. That's fine, but learning how to run Python terminal is still a useful skill to know.

### Interactive Mode

When you start Python, you get an interactive mode where you can experiment.

If you start typing statements, they will run immediately. There is no edit/compile/run/debug cycle.

>>> print('hello world')

hello world

>>> 37\*42

1554

>>> for i in range(5):

... print(i)

...

0

1

2

3

4

>>>

This so-called read-eval-print-loop (or REPL) is very useful for debugging and exploration.

**STOP**: If you can't figure out how to interact with Python, stop what you're doing and figure out how to do it. If you're using an IDE, it might be hidden behind a menu option or other window. Many parts of this course assume that you can interact with the interpreter.

Let's take a closer look at the elements of the REPL:

* >>> is the interpreter prompt for starting a new statement.
* ... is the interpreter prompt for continuing a statement. Enter a blank line to finish typing and run what you've entered.

The ... prompt may or may not be shown depending on your environment. For this course, it is shown as blanks to make it easier to cut/paste code samples.

The underscore \_ holds the last result.

>>> 37 \* 42

1554

>>> \_ \* 2

3108

>>> \_ + 50

3158

>>>

This is only true in the interactive mode. You never use \_ in a program.

### Creating programs

Programs are put in .py files.

# hello.py

print('hello world')

You can create these files with your favorite text editor.

### Running Programs

To execute a program, run it in the terminal with the python command. For example, in command-line Unix:

bash % python hello.py

hello world

bash %

Or from the Windows shell:

C:\SomeFolder>hello.py

hello world

C:\SomeFolder>c:\python36\python hello.py

hello world

Note: On Windows, you may need to specify a full path to the Python interpreter such as c:\python36\python. However, if Python is installed in its usual way, you might be able to just type the name of the program such as hello.py.

### A Sample Program

Let's solve the following problem:

One morning, you go out and place a dollar bill on the sidewalk by the Sears tower in Chicago. Each day thereafter, you go out double the number of bills. How long does it take for the stack of bills to exceed the height of the tower?

Here's a solution:

# sears.py

bill\_thickness = 0.11 \* 0.001 # Meters (0.11 mm)

sears\_height = 442 # Height (meters)

num\_bills = 1

day = 1

while num\_bills \* bill\_thickness < sears\_height:

print(day, num\_bills, num\_bills \* bill\_thickness)

day = day + 1

num\_bills = num\_bills \* 2

print('Number of days', day)

print('Number of bills', num\_bills)

print('Final height', num\_bills \* bill\_thickness)

When you run it, you get the following output:

bash % python3 sears.py

1 1 0.00011

2 2 0.00022

3 4 0.00044

4 8 0.00088

5 16 0.00176

6 32 0.00352

...

21 1048576 115.34336

22 2097152 230.68672

Number of days 23

Number of bills 4194304

Final height 461.37344

Using this program as a guide, you can learn a number of important core concepts about Python.

### Statements

A python program is a sequence of statements:

a = 3 + 4

b = a \* 2

print(b)

Each statement is terminated by a newline. Statements are executed one after the other until control reaches the end of the file.

### Comments

Comments are text that will not be executed.

a = 3 + 4

# This is a comment

b = a \* 2

print(b)

Comments are denoted by # and extend to the end of the line.

### Variables

A variable is a name for a value. You can use letters (lower and upper-case) from a to z. As well as the character underscore \_. Numbers can also be part of the name of a variable, except as the first character.

height = 442 # valid

\_height = 442 # valid

height2 = 442 # valid

2height = 442 # invalid

### Types

Variables do not need to be declared with the type of the value. The type is associated with the value on the right hand side, not name of the variable.

height = 442 # An integer

height = 442.0 # Floating point

height = 'Really tall' # A string

Python is dynamically typed. The perceived "type" of a variable might change as a program executes depending on the current value assigned to it.

### Case Sensitivity

Python is case sensitive. Upper and lower-case letters are considered different letters. These are all different variables:

name = 'Jake'

Name = 'Elwood'

NAME = 'Guido'

Language statements are always lower-case.

while x < 0: # OK

WHILE x < 0: # ERROR

### Looping

The while statement executes a loop.

while num\_bills \* bill\_thickness < sears\_height:

print(day, num\_bills, num\_bills \* bill\_thickness)

day = day + 1

num\_bills = num\_bills \* 2

print('Number of days', day)

The statements indented below the while will execute as long as the expression after the while is true.

### Indentation

Indentation is used to denote groups of statements that go together. Consider the previous example:

while num\_bills \* bill\_thickness < sears\_height:

print(day, num\_bills, num\_bills \* bill\_thickness)

day = day + 1

num\_bills = num\_bills \* 2

print('Number of days', day)

Indentation groups the following statements together as the operations that repeat:

print(day, num\_bills, num\_bills \* bill\_thickness)

day = day + 1

num\_bills = num\_bills \* 2

Because the print() statement at the end is not indented, it does not belong to the loop. The empty line is just for readability. It does not affect the execution.

### Indentation best practices

* Use spaces instead of tabs.
* Use 4 spaces per level.
* Use a Python-aware editor.

Python's only requirement is that indentation within the same block be consistent. For example, this is an error:

while num\_bills \* bill\_thickness < sears\_height:

print(day, num\_bills, num\_bills \* bill\_thickness)

day = day + 1 # ERROR

num\_bills = num\_bills \* 2

### Conditionals

The if statement is used to execute a conditional:

if a > b:

print('Computer says no')

else:

print('Computer says yes')

You can check for multiple conditions by adding extra checks using elif.

if a > b:

print('Computer says no')

elif a == b:

print('Computer says yes')

else:

print('Computer says maybe')

### Printing

The print function produces a single line of text with the values passed.

print('Hello world!') # Prints the text 'Hello world!'

You can use variables. The text printed will be the value of the variable, not the name.

x = 100

print(x) # Prints the text '100'

If you pass more than one value to print they are separated by spaces.

name = 'Jake'

print('My name is', name) # Print the text 'My name is Jake'

print() always puts a newline at the end.

print('Hello')

print('My name is', 'Jake')

This prints:

Hello

My name is Jake

The extra newline can be suppressed:

print('Hello', end=' ')

print('My name is', 'Jake')

This code will now print:

Hello My name is Jake

### User input

To read a line of typed user input, use the input() function:

name = input('Enter your name:')

print('Your name is', name)

input prints a prompt to the user and returns their response. This is useful for small programs, learning exercises or simple debugging. It is not widely used for real programs.

### pass statement

Sometimes you need to specify an empty code block. The keyword pass is used for it.

if a > b:

pass

else:

print('Computer says false')

This is also called a "no-op" statement. It does nothing. It serves as a placeholder for statements, possibly to be added later.

## Exercises

This is the first set of exercises where you need to create Python files and run them. From this point forward, it is assumed that you are editing files in the practical-python/Work/ directory. To help you locate the proper place, a number of empty starter files have been created with the appropriate filenames. Look for the file Work/bounce.py that's used in the first exercise.

### Exercise 1.5: The Bouncing Ball

A rubber ball is dropped from a height of 100 meters and each time it hits the ground, it bounces back up to 3/5 the height it fell. Write a program bounce.py that prints a table showing the height of the first 10 bounces.

Your program should make a table that looks something like this:

1 60.0

2 36.0

3 21.599999999999998

4 12.959999999999999

5 7.775999999999999

6 4.6655999999999995

7 2.7993599999999996

8 1.6796159999999998

9 1.0077695999999998

10 0.6046617599999998

Note: You can clean up the output a bit if you use the round() function. Try using it to round the output to 4 digits.

1 60.0

2 36.0

3 21.6

4 12.96

5 7.776

6 4.6656

7 2.7994

8 1.6796

9 1.0078

10 0.6047

### Exercise 1.6: Debugging

The following code fragment contains code from the Sears tower problem. It also has a bug in it.

# sears.py

bill\_thickness = 0.11 \* 0.001 # Meters (0.11 mm)

sears\_height = 442 # Height (meters)

num\_bills = 1

day = 1

while num\_bills \* bill\_thickness < sears\_height:

print(day, num\_bills, num\_bills \* bill\_thickness)

day = days + 1

num\_bills = num\_bills \* 2

print('Number of days', day)

print('Number of bills', num\_bills)

print('Final height', num\_bills \* bill\_thickness)

Copy and paste the code that appears above in a new program called sears.py. When you run the code you will get an error message that causes the program to crash like this:

Traceback (most recent call last):

File "sears.py", line 10, in <module>

day = days + 1

NameError: name 'days' is not defined

Reading error messages is an important part of Python code. If your program crashes, the very last line of the traceback message is the actual reason why the the program crashed. Above that, you should see a fragment of source code and then an identifying filename and line number.

* Which line is the error?
* What is the error?
* Fix the error
* Run the program successfully

[Contents](https://github.com/ashkan-abbasi66/python-comments/blob/master/prog/dabeaz/Notes/Contents.md) | [Previous (1.1 Python)](https://github.com/ashkan-abbasi66/python-comments/blob/master/prog/dabeaz/Notes/01_Introduction/01_Python.md) | [Next (1.3 Numbers)](https://github.com/ashkan-abbasi66/python-comments/blob/master/prog/dabeaz/Notes/01_Introduction/03_Numbers.md)

# .3 Numbers

This section discusses mathematical calculations.

### Types of Numbers

Python has 4 types of numbers:

* Booleans
* Integers
* Floating point
* Complex (imaginary numbers)

### Booleans (bool)

Booleans have two values: True, False.

a = True

b = False

Numerically, they're evaluated as integers with value 1, 0.

c = 4 + True # 5

d = False

if d == 0:

print('d is False')

But, don't write code like that. It would be odd.

### Integers (int)

Signed values of arbitrary size and base:

a = 37

b = -299392993727716627377128481812241231

c = 0x7fa8 # Hexadecimal

d = 0o253 # Octal

e = 0b10001111 # Binary

Common operations:

x + y Add

x - y Subtract

x \* y Multiply

x / y Divide (produces a float)

x // y Floor Divide (produces an integer)

x % y Modulo (remainder)

x \*\* y Power

x << n Bit shift left

x >> n Bit shift right

x & y Bit-wise AND

x | y Bit-wise OR

x ^ y Bit-wise XOR

~x Bit-wise NOT

abs(x) Absolute value

### Floating point (float)

Use a decimal or exponential notation to specify a floating point value:

a = 37.45

b = 4e5 # 4 x 10\*\*5 or 400,000

c = -1.345e-10

Floats are represented as double precision using the native CPU representation [IEEE 754](https://en.wikipedia.org/wiki/IEEE_754). This is the same as the double type in the programming language C.

17 digits of precision  
Exponent from -308 to 308

Be aware that floating point numbers are inexact when representing decimals.

>>> a = 2.1 + 4.2

>>> a == 6.3

False

>>> a

6.300000000000001

>>>

This is **not a Python issue**, but the underlying floating point hardware on the CPU.

Common Operations:

x + y Add

x - y Subtract

x \* y Multiply

x / y Divide

x // y Floor Divide

x % y Modulo

x \*\* y Power

abs(x) Absolute Value

These are the same operators as Integers, except for the bit-wise operators. Additional math functions are found in the math module.

import math

a = math.sqrt(x)

b = math.sin(x)

c = math.cos(x)

d = math.tan(x)

e = math.log(x)

### Comparisons

The following comparison / relational operators work with numbers:

x < y Less than

x <= y Less than or equal

x > y Greater than

x >= y Greater than or equal

x == y Equal to

x != y Not equal to

You can form more complex boolean expressions using

and, or, not

Here are a few examples:

if b >= a and b <= c:

print('b is between a and c')

if not (b < a or b > c):

print('b is still between a and c')

### Converting Numbers

The type name can be used to convert values:

a = int(x) # Convert x to integer

b = float(x) # Convert x to float

Try it out.

>>> a = 3.14159

>>> int(a)

3

>>> b = '3.14159' # It also works with strings containing numbers

>>> float(b)

3.14159

>>>

## Exercises

Reminder: These exercises assume you are working in the practical-python/Work directory. Look for the file mortgage.py.

### Exercise 1.7: Dave's mortgage

Dave has decided to take out a 30-year fixed rate mortgage of $500,000 with Guido’s Mortgage, Stock Investment, and Bitcoin trading corporation. The interest rate is 5% and the monthly payment is $2684.11.

Here is a program that calculates the total amount that Dave will have to pay over the life of the mortgage:

# mortgage.py

principal = 500000.0

rate = 0.05

payment = 2684.11

total\_paid = 0.0

while principal > 0:

principal = principal \* (1+rate/12) - payment

total\_paid = total\_paid + payment

print('Total paid', total\_paid)

Enter this program and run it. You should get an answer of 966,279.6.

### Exercise 1.8: Extra payments

Suppose Dave pays an extra $1000/month for the first 12 months of the mortgage?

Modify the program to incorporate this extra payment and have it print the total amount paid along with the number of months required.

When you run the new program, it should report a total payment of 929,965.62 over 342 months.

### Exercise 1.9: Making an Extra Payment Calculator

Modify the program so that extra payment information can be more generally handled. Make it so that the user can set these variables:

extra\_payment\_start\_month = 61

extra\_payment\_end\_month = 108

extra\_payment = 1000

Make the program look at these variables and calculate the total paid appropriately.

How much will Dave pay if he pays an extra $1000/month for 4 years starting after the first five years have already been paid?

### Exercise 1.10: Making a table

Modify the program to print out a table showing the month, total paid so far, and the remaining principal. The output should look something like this:

1 2684.11 499399.22

2 5368.22 498795.94

3 8052.33 498190.15

4 10736.44 497581.83

5 13420.55 496970.98

...

308 874705.88 3478.83

309 877389.99 809.21

310 880074.1 -1871.53

Total paid 880074.1

Months 310

### Exercise 1.11: Bonus

While you’re at it, fix the program to correct for the overpayment that occurs in the last month.

### Exercise 1.12: A Mystery

int() and float() can be used to convert numbers. For example,

>>> int("123")

123

>>> float("1.23")

1.23

>>>

With that in mind, can you explain this behavior?

>>> bool("False")

True

>>>

[Contents](https://github.com/ashkan-abbasi66/python-comments/blob/master/prog/dabeaz/Notes/Contents.md) | [Previous (1.2 A First Program)](https://github.com/ashkan-abbasi66/python-comments/blob/master/prog/dabeaz/Notes/01_Introduction/02_Hello_world.md) | [Next (1.4 Strings)](https://github.com/ashkan-abbasi66/python-comments/blob/master/prog/dabeaz/Notes/01_Introduction/04_Strings.md)

# 1.4 Strings

This section introduces ways to work with text.

### Representing Literal Text

String literals are written in programs with quotes.

# Single quote

a = 'Yeah but no but yeah but...'

# Double quote

b = "computer says no"

# Triple quotes

c = '''

Look into my eyes, look into my eyes, the eyes, the eyes, the eyes,

not around the eyes,

don't look around the eyes,

look into my eyes, you're under.

'''

Normally strings may only span a single line. Triple quotes capture all text enclosed across multiple lines including all formatting.

There is no difference between using single (') versus double (") quotes. However, the same type of quote used to start a string must be used to terminate it.

### String escape codes

Escape codes are used to represent control characters and characters that can't be easily typed directly at the keyboard. Here are some common escape codes:

'\n' Line feed

'\r' Carriage return

'\t' Tab

'\'' Literal single quote

'\"' Literal double quote

'\\' Literal backslash

### String Representation

Each character in a string is stored internally as a so-called Unicode "code-point" which is an integer. You can specify an exact code-point value using the following escape sequences:

a = '\xf1' # a = 'ñ'

b = '\u2200' # b = '∀'

c = '\U0001D122' # c = '𝄢'

d = '\N{FOR ALL}' # d = '∀'

The [Unicode Character Database](https://unicode.org/charts) is a reference for all available character codes.

### String Indexing

Strings work like an array for accessing individual characters. You use an integer index, starting at 0. Negative indices specify a position relative to the end of the string.

a = 'Hello world'

b = a[0] # 'H'

c = a[4] # 'o'

d = a[-1] # 'd' (end of string)

You can also slice or select substrings specifying a range of indices with :.

d = a[:5] # 'Hello'

e = a[6:] # 'world'

f = a[3:8] # 'lo wo'

g = a[-5:] # 'world'

The character at the ending index is not included. Missing indices assume the beginning or ending of the string.

### String operations

Concatenation, length, membership and replication.

# Concatenation (+)

a = 'Hello' + 'World' # 'HelloWorld'

b = 'Say ' + a # 'Say HelloWorld'

# Length (len)

s = 'Hello'

len(s) # 5

# Membership test (`in`, `not in`)

t = 'e' in s # True

f = 'x' in s # False

g = 'hi' not in s # True

# Replication (s \* n)

rep = s \* 5 # 'HelloHelloHelloHelloHello'

### String methods

Strings have methods that perform various operations with the string data.

Example: stripping any leading / trailing white space.

s = ' Hello '

t = s.strip() # 'Hello'

Example: Case conversion.

s = 'Hello'

l = s.lower() # 'hello'

u = s.upper() # 'HELLO'

Example: Replacing text.

s = 'Hello world'

t = s.replace('Hello' , 'Hallo') # 'Hallo world'

**More string methods:**

Strings have a wide variety of other methods for testing and manipulating the text data. This is a small sample of methods:

s.endswith(suffix) # Check if string ends with suffix

s.find(t) # First occurrence of t in s

s.index(t) # First occurrence of t in s

s.isalpha() # Check if characters are alphabetic

s.isdigit() # Check if characters are numeric

s.islower() # Check if characters are lower-case

s.isupper() # Check if characters are upper-case

s.join(slist) # Join a list of strings using s as delimiter

s.lower() # Convert to lower case

s.replace(old,new) # Replace text

s.rfind(t) # Search for t from end of string

s.rindex(t) # Search for t from end of string

s.split([delim]) # Split string into list of substrings

s.startswith(prefix) # Check if string starts with prefix

s.strip() # Strip leading/trailing space

s.upper() # Convert to upper case

### String Mutability

Strings are "immutable" or read-only. Once created, the value can't be changed.

>>> s = 'Hello World'

>>> s[1] = 'a'

Traceback (most recent call last):

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

TypeError: 'str' object does not support item assignment

>>>

**All operations and methods that manipulate string data, always create new strings.**

### String Conversions

Use str() to convert any value to a string. The result is a string holding the same text that would have been produced by the print() statement.

>>> x = 42

>>> str(x)

'42'

>>>

### Byte Strings

A string of 8-bit bytes, commonly encountered with low-level I/O, is written as follows:

data = b'Hello World\r\n'

By putting a little b before the first quotation, you specify that it is a byte string as opposed to a text string.

Most of the usual string operations work.

len(data) # 13

data[0:5] # b'Hello'

data.replace(b'Hello', b'Cruel') # b'Cruel World\r\n'

Indexing is a bit different because it returns byte values as integers.

data[0] # 72 (ASCII code for 'H')

Conversion to/from text strings.

text = data.decode('utf-8') # bytes -> text

data = text.encode('utf-8') # text -> bytes

The 'utf-8' argument specifies a character encoding. Other common values include 'ascii' and 'latin1'.

### Raw Strings

Raw strings are string literals with an uninterpreted backslash. They are specified by prefixing the initial quote with a lowercase "r".

>>> rs = r'c:\newdata\test' # Raw (uninterpreted backslash)

>>> rs

'c:\\newdata\\test'

The string is the literal text enclosed inside, exactly as typed. This is useful in situations where the backslash has special significance. Example: filename, regular expressions, etc.

### f-Strings

A string with formatted expression substitution.

>>> name = 'IBM'

>>> shares = 100

>>> price = 91.1

>>> a = f'{name:>10s} {shares:10d} {price:10.2f}'

>>> a

' IBM 100 91.10'

>>> b = f'Cost = ${shares\*price:0.2f}'

>>> b

'Cost = $9110.00'

>>>

**Note: This requires Python 3.6 or newer.** The meaning of the format codes is covered later.

## Exercises

In these exercises, you'll experiment with operations on Python's string type. You should do this at the Python interactive prompt where you can easily see the results. Important note:

In exercises where you are supposed to interact with the interpreter, >>> is the interpreter prompt that you get when Python wants you to type a new statement. Some statements in the exercise span multiple lines--to get these statements to run, you may have to hit 'return' a few times. Just a reminder that you DO NOT type the >>> when working these examples.

Start by defining a string containing a series of stock ticker symbols like this:

>>> symbols = 'AAPL,IBM,MSFT,YHOO,SCO'

>>>

### Exercise 1.13: Extracting individual characters and substrings

Strings are arrays of characters. Try extracting a few characters:

>>> symbols[0]

?

>>> symbols[1]

?

>>> symbols[2]

?

>>> symbols[-1] # Last character

?

>>> symbols[-2] # Negative indices are from end of string

?

>>>

In Python, strings are read-only.

Verify this by trying to change the first character of symbols to a lower-case 'a'.

>>> symbols[0] = 'a'

Traceback (most recent call last):

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

TypeError: 'str' object does not support item assignment

>>>

### Exercise 1.14: String concatenation

Although string data is read-only, you can always reassign a variable to a newly created string.

Try the following statement which concatenates a new symbol "GOOG" to the end of symbols:

>>> symbols = symbols + 'GOOG'

>>> symbols

'AAPL,IBM,MSFT,YHOO,SCOGOOG'

>>>

Oops! That's not what you wanted. Fix it so that the symbols variable holds the value 'AAPL,IBM,MSFT,YHOO,SCO,GOOG'.

>>> symbols = ?

>>> symbols

'AAPL,IBM,MSFT,YHOO,SCO,GOOG'

>>>

Add 'HPQ' to the front the string:

>>> symbols = ?

>>> symbols

'HPQ,AAPL,IBM,MSFT,YHOO,SCO,GOOG'

>>>

In these examples, it might look like the original string is being modified, in an apparent violation of strings being read only. Not so. Operations on strings create an entirely new string each time. When the variable name symbols is reassigned, it points to the newly created string. Afterwards, the old string is destroyed since it's not being used anymore.

### Exercise 1.15: Membership testing (substring testing)

Experiment with the in operator to check for substrings. At the interactive prompt, try these operations:

>>> 'IBM' in symbols

?

>>> 'AA' in symbols

True

>>> 'CAT' in symbols

?

>>>

Why did the check for *'AA'* return *True*?

### Exercise 1.16: String Methods

At the Python interactive prompt, try experimenting with some of the string methods.

>>> symbols.lower()

?

>>> symbols

?

>>>

Remember, strings are always read-only. If you want to save the result of an operation, you need to place it in a variable:

>>> lowersyms = symbols.lower()

>>>

Try some more operations:

>>> symbols.find('MSFT')

?

>>> symbols[13:17]

?

>>> symbols = symbols.replace('SCO','DOA')

>>> symbols

?

>>> name = ' IBM \n'

>>> name = name.strip() # Remove surrounding whitespace

>>> name

?

>>>

### Exercise 1.17: f-strings

Sometimes you want to create a string and embed the values of variables into it.

To do that, use an f-string. For example:

>>> name = 'IBM'

>>> shares = 100

>>> price = 91.1

>>> f'{shares} shares of {name} at ${price:0.2f}'

'100 shares of IBM at $91.10'

>>>

Modify the mortgage.py program from [Exercise 1.10](https://github.com/ashkan-abbasi66/python-comments/blob/master/prog/dabeaz/Notes/01_Introduction/03_Numbers.md) to create its output using f-strings. Try to make it so that output is nicely aligned.

### Exercise 1.18: Regular Expressions

One limitation of the basic string operations is that they don't support any kind of advanced pattern matching. For that, you need to turn to Python's re module and regular expressions. Regular expression handling is a big topic, but here is a short example:

>>> text = 'Today is 3/27/2018. Tomorrow is 3/28/2018.'

>>> # Find all occurrences of a date

>>> import re

>>> re.findall(r'\d+/\d+/\d+', text)

['3/27/2018', '3/28/2018']

>>> # Replace all occurrences of a date with replacement text

>>> re.sub(r'(\d+)/(\d+)/(\d+)', r'\3-\1-\2', text)

'Today is 2018-3-27. Tomorrow is 2018-3-28.'

>>>

For more information about the re module, see the official documentation at [https://docs.python.org/library/re.html](https://docs.python.org/3/library/re.html).

### Commentary

As you start to experiment with the interpreter, you often want to know more about the operations supported by different objects. For example, how do you find out what operations are available on a string?

Depending on your Python environment, you might be able to see a list of available methods via tab-completion. For example, try typing this:

>>> s = 'hello world'

>>> s.<tab key>

>>>

If hitting tab doesn't do anything, you can fall back to the builtin-in dir() function. For example:

>>> s = 'hello'

>>> dir(s)

['\_\_add\_\_', '\_\_class\_\_', '\_\_contains\_\_', ..., 'find', 'format',

'index', 'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace',

'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'partition',

'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit',

'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase',

'title', 'translate', 'upper', 'zfill']

>>>

dir() produces a list of all operations that can appear after the (.). Use the help() command to get more information about a specific operation:

>>> help(s.upper)

Help on built-in function upper:

upper(...)

S.upper() -> string

Return a copy of the string S converted to uppercase.

>>>

[Contents](https://github.com/ashkan-abbasi66/python-comments/blob/master/prog/dabeaz/Notes/Contents.md) | [Previous (1.3 Numbers)](https://github.com/ashkan-abbasi66/python-comments/blob/master/prog/dabeaz/Notes/01_Introduction/03_Numbers.md) | [Next (1.5 Lists)](https://github.com/ashkan-abbasi66/python-comments/blob/master/prog/dabeaz/Notes/01_Introduction/05_Lists.md)

# 1.5 Lists

This section introduces lists, Python's primary type for holding an ordered collection of values.

### Creating a List

Use square brackets to define a list literal:

names = [ 'Elwood', 'Jake', 'Curtis' ]

nums = [ 39, 38, 42, 65, 111]

Sometimes lists are created by other methods. For example, a string can be split into a list using the split() method:

>>> line = 'GOOG,100,490.10'

>>> row = line.split(',')

>>> row

['GOOG', '100', '490.10']

>>>

### List operations

Lists can hold items of any type. Add a new item using append():

names.append('Murphy') # Adds at end

names.insert(2, 'Aretha') # Inserts in middle

Use + to concatenate lists:

s = [1, 2, 3]

t = ['a', 'b']

s + t # [1, 2, 3, 'a', 'b']

Lists are indexed by integers. Starting at 0.

names = [ 'Elwood', 'Jake', 'Curtis' ]

names[0] # 'Elwood'

names[1] # 'Jake'

names[2] # 'Curtis'

Negative indices count from the end.

names[-1] # 'Curtis'

You can change any item in a list.

names[1] = 'Joliet Jake'

names # [ 'Elwood', 'Joliet Jake', 'Curtis' ]

Length of the list.

names = ['Elwood','Jake','Curtis']

len(names) # 3

Membership test (in, not in).

'Elwood' in names # True

'Britney' not in names # True

Replication (s \* n).

s = [1, 2, 3]

s \* 3 # [1, 2, 3, 1, 2, 3, 1, 2, 3]

### List Iteration and Search

Use for to iterate over the list contents.

for name in names:

# use name

# e.g. print(name)

...

This is similar to a foreach statement from other programming languages.

To find the position of something quickly, use index().

names = ['Elwood','Jake','Curtis']

names.index('Curtis') # 2

If the element is present more than once, index() will return the index of the first occurrence.

If the element is not found, it will raise a ValueError exception.

### List Removal

You can remove items either by element value or by index:

# Using the value

names.remove('Curtis')

# Using the index

del names[1]

Removing an item does not create a hole. Other items will move down to fill the space vacated. If there are more than one occurrence of the element, remove() will remove only the first occurrence.

### List Sorting

Lists can be sorted "in-place".

s = [10, 1, 7, 3]

s.sort() # [1, 3, 7, 10]

# Reverse order

s = [10, 1, 7, 3]

s.sort(reverse=True) # [10, 7, 3, 1]

# It works with any ordered data

s = ['foo', 'bar', 'spam']

s.sort() # ['bar', 'foo', 'spam']

Use sorted() if you'd like to make a new list instead:

t = sorted(s) # s unchanged, t holds sorted values

### Lists and Math

Caution: Lists were not designed for math operations.

>>> nums = [1, 2, 3, 4, 5]

>>> nums \* 2

[1, 2, 3, 4, 5, 1, 2, 3, 4, 5]

>>> nums + [10, 11, 12, 13, 14]

[1, 2, 3, 4, 5, 10, 11, 12, 13, 14]

Specifically, lists don't represent vectors/matrices as in MATLAB, Octave, R, etc. However, there are some packages to help you with that (e.g. [numpy](https://numpy.org/)).

## Exercises

In this exercise, we experiment with Python's list datatype. In the last section, you worked with strings containing stock symbols.

>>> symbols = 'HPQ,AAPL,IBM,MSFT,YHOO,DOA,GOOG'

Split it into a list of names using the split() operation of strings:

>>> symlist = symbols.split(',')

### Exercise 1.19: Extracting and reassigning list elements

Try a few lookups:

>>> symlist[0]

'HPQ'

>>> symlist[1]

'AAPL'

>>> symlist[-1]

'GOOG'

>>> symlist[-2]

'DOA'

>>>

Try reassigning one value:

>>> symlist[2] = 'AIG'

>>> symlist

['HPQ', 'AAPL', 'AIG', 'MSFT', 'YHOO', 'DOA', 'GOOG']

>>>

Take a few slices:

>>> symlist[0:3]

['HPQ', 'AAPL', 'AIG']

>>> symlist[-2:]

['DOA', 'GOOG']

>>>

Create an empty list and append an item to it.

>>> mysyms = []

>>> mysyms.append('GOOG')

>>> mysyms

['GOOG']

You can reassign a portion of a list to another list. For example:

>>> symlist[-2:] = mysyms

>>> symlist

['HPQ', 'AAPL', 'AIG', 'MSFT', 'YHOO', 'GOOG']

>>>

When you do this, the list on the left-hand-side (symlist) will be resized as appropriate to make the right-hand-side (mysyms) fit. For instance, in the above example, the last two items of symlist got replaced by the single item in the list mysyms.

### Exercise 1.20: Looping over list items

The for loop works by looping over data in a sequence such as a list. Check this out by typing the following loop and watching what happens:

>>> for s in symlist:

print('s =', s)

# Look at the output

### Exercise 1.21: Membership tests

Use the in or not in operator to check if 'AIG','AA', and 'CAT' are in the list of symbols.

>>> # Is 'AIG' IN the `symlist`?

True

>>> # Is 'AA' IN the `symlist`?

False

>>> # Is 'CAT' NOT IN the `symlist`?

True

>>>

### Exercise 1.22: Appending, inserting, and deleting items

Use the append() method to add the symbol 'RHT' to end of symlist.

>>> # append 'RHT'

>>> symlist

['HPQ', 'AAPL', 'AIG', 'MSFT', 'YHOO', 'GOOG', 'RHT']

>>>

Use the insert() method to insert the symbol 'AA' as the second item in the list.

>>> # Insert 'AA' as the second item in the list

>>> symlist

['HPQ', 'AA', 'AAPL', 'AIG', 'MSFT', 'YHOO', 'GOOG', 'RHT']

>>>

Use the remove() method to remove 'MSFT' from the list.

>>> # Remove 'MSFT'

>>> symlist

['HPQ', 'AA', 'AAPL', 'AIG', 'YHOO', 'GOOG', 'RHT']

>>>

Append a duplicate entry for 'YHOO' at the end of the list.

Note: it is perfectly fine for a list to have duplicate values.

>>> # Append 'YHOO'

>>> symlist

['HPQ', 'AA', 'AAPL', 'AIG', 'YHOO', 'GOOG', 'RHT', 'YHOO']

>>>

Use the index() method to find the first position of 'YHOO' in the list.

>>> # Find the first index of 'YHOO'

4

>>> symlist[4]

'YHOO'

>>>

Count how many times 'YHOO' is in the list:

>>> symlist.count('YHOO')

2

>>>

Remove the first occurrence of 'YHOO'.

>>> # Remove first occurrence 'YHOO'

>>> symlist

['HPQ', 'AA', 'AAPL', 'AIG', 'GOOG', 'RHT', 'YHOO']

>>>

Just so you know, there is no method to find or remove all occurrences of an item. However, we'll see an elegant way to do this in section 2.

### Exercise 1.23: Sorting

Want to sort a list? Use the sort() method. Try it out:

>>> symlist.sort()

>>> symlist

['AA', 'AAPL', 'AIG', 'GOOG', 'HPQ', 'RHT', 'YHOO']

>>>

Want to sort in reverse? Try this:

>>> symlist.sort(reverse=True)

>>> symlist

['YHOO', 'RHT', 'HPQ', 'GOOG', 'AIG', 'AAPL', 'AA']

>>>

Note: Sorting a list modifies its contents 'in-place'. That is, the elements of the list are shuffled around, but no new list is created as a result.

### Exercise 1.24: Putting it all back together

Want to take a list of strings and join them together into one string? Use the join() method of strings like this (note: this looks funny at first).

>>> a = ','.join(symlist)

>>> a

'YHOO,RHT,HPQ,GOOG,AIG,AAPL,AA'

>>> b = ':'.join(symlist)

>>> b

'YHOO:RHT:HPQ:GOOG:AIG:AAPL:AA'

>>> c = ''.join(symlist)

>>> c

'YHOORHTHPQGOOGAIGAAPLAA'

>>>

### Exercise 1.25: Lists of anything

Lists can contain any kind of object, including other lists (e.g., nested lists). Try this out:

>>> nums = [101, 102, 103]

>>> items = ['spam', symlist, nums]

>>> items

['spam', ['YHOO', 'RHT', 'HPQ', 'GOOG', 'AIG', 'AAPL', 'AA'], [101, 102, 103]]

Pay close attention to the above output. items is a list with three elements. The first element is a string, but the other two elements are lists.

You can access items in the nested lists by using multiple indexing operations.

>>> items[0]

'spam'

>>> items[0][0]

's'

>>> items[1]

['YHOO', 'RHT', 'HPQ', 'GOOG', 'AIG', 'AAPL', 'AA']

>>> items[1][1]

'RHT'

>>> items[1][1][2]

'T'

>>> items[2]

[101, 102, 103]

>>> items[2][1]

102

>>>

Even though it is technically possible to make very complicated list structures, as a general rule, you want to keep things simple. Usually lists hold items that are all the same kind of value. For example, a list that consists entirely of numbers or a list of text strings. Mixing different kinds of data together in the same list is often a good way to make your head explode so it's best avoided.

[Contents](https://github.com/ashkan-abbasi66/python-comments/blob/master/prog/dabeaz/Notes/Contents.md) | [Previous (1.4 Strings)](https://github.com/ashkan-abbasi66/python-comments/blob/master/prog/dabeaz/Notes/01_Introduction/04_Strings.md) | [Next (1.6 Files)](https://github.com/ashkan-abbasi66/python-comments/blob/master/prog/dabeaz/Notes/01_Introduction/06_Files.md)

# 1.6 File Management

Most programs need to read input from somewhere. This section discusses file access.

### File Input and Output

Open a file.

f = open('foo.txt', 'rt') # Open for reading (text)

g = open('bar.txt', 'wt') # Open for writing (text)

Read all of the data.

data = f.read()

# Read only up to 'maxbytes' bytes

data = f.read([maxbytes])

Write some text.

g.write('some text')

Close when you are done.

f.close()

g.close()

Files should be properly closed and it's an easy step to forget. Thus, the preferred approach is to use the with statement like this.

with open(filename, 'rt') as file:

# Use the file `file`

...

# No need to close explicitly

...statements

This automatically closes the file when control leaves the indented code block.

### Common Idioms for Reading File Data

Read an entire file all at once as a string.

with open('foo.txt', 'rt') as file:

data = file.read()

# `data` is a string with all the text in `foo.txt`

Read a file line-by-line by iterating.

with open(filename, 'rt') as file:

for line in file:

# Process the line

### Common Idioms for Writing to a File

Write string data.

with open('outfile', 'wt') as out:

out.write('Hello World\n')

...

Redirect the print function.

with open('outfile', 'wt') as out:

print('Hello World', file=out)

...

## Exercises

These exercises depend on a file Data/portfolio.csv. The file contains a list of lines with information on a portfolio of stocks. It is assumed that you are working in the practical-python/Work/ directory. If you're not sure, you can find out where Python thinks it's running by doing this:

>>> import os

>>> os.getcwd()

'/Users/beazley/Desktop/practical-python/Work' # Output vary

>>>

### Exercise 1.26: File Preliminaries

First, try reading the entire file all at once as a big string:

>>> with open('Data/portfolio.csv', 'rt') as f:

data = f.read()

>>> data

'name,shares,price\n"AA",100,32.20\n"IBM",50,91.10\n"CAT",150,83.44\n"MSFT",200,51.23\n"GE",95,40.37\n"MSFT",50,65.10\n"IBM",100,70.44\n'

>>> print(data)

name,shares,price

"AA",100,32.20

"IBM",50,91.10

"CAT",150,83.44

"MSFT",200,51.23

"GE",95,40.37

"MSFT",50,65.10

"IBM",100,70.44

>>>

In the above example, it should be noted that Python has two modes of output. In the first mode where you type data at the prompt, Python shows you the raw string representation including quotes and escape codes. When you type print(data), you get the actual formatted output of the string.

Although reading a file all at once is simple, it is often not the most appropriate way to do it—especially if the file happens to be huge or if contains lines of text that you want to handle one at a time.

To read a file line-by-line, use a for-loop like this:

>>> with open('Data/portfolio.csv', 'rt') as f:

for line in f:

print(line, end='')

name,shares,price

"AA",100,32.20

"IBM",50,91.10

...

>>>

When you use this code as shown, lines are read until the end of the file is reached at which point the loop stops.

On certain occasions, you might want to manually read or skip a single line of text (e.g., perhaps you want to skip the first line of column headers).

>>> f = open('Data/portfolio.csv', 'rt')

>>> headers = next(f)

>>> headers

'name,shares,price\n'

>>> for line in f:

print(line, end='')

"AA",100,32.20

"IBM",50,91.10

...

>>> f.close()

>>>

next() returns the next line of text in the file. If you were to call it repeatedly, you would get successive lines. However, just so you know, the for loop already uses next() to obtain its data. Thus, you normally wouldn’t call it directly unless you’re trying to explicitly skip or read a single line as shown.

Once you’re reading lines of a file, you can start to perform more processing such as splitting. For example, try this:

>>> f = open('Data/portfolio.csv', 'rt')

>>> headers = next(f).split(',')

>>> headers

['name', 'shares', 'price\n']

>>> for line in f:

row = line.split(',')

print(row)

['"AA"', '100', '32.20\n']

['"IBM"', '50', '91.10\n']

...

>>> f.close()

Note: In these examples, *f.close()* is being called explicitly because the *with* statement isn’t being used.

### Exercise 1.27: Reading a data file

Now that you know how to read a file, let’s write a program to perform a simple calculation.

The columns in portfolio.csv correspond to the stock name, number of shares, and purchase price of a single stock holding. Write a program called pcost.py that opens this file, reads all lines, and calculates how much it cost to purchase all of the shares in the portfolio.

Hint: to convert a string to an integer, use *int(s)*. To convert a string to a floating point, use *float(s)*.

Your program should print output such as the following:

Total cost 44671.15

### Exercise 1.28: Other kinds of "files"

What if you wanted to read a non-text file such as a gzip-compressed datafile? The builtin open() function won’t help you here, but Python has a library module gzip that can read gzip compressed files.

Try it:

>>> import gzip

>>> with gzip.open('Data/portfolio.csv.gz', 'rt') as f:

for line in f:

print(line, end='')

... look at the output ...

>>>

Note: Including the file mode of 'rt' is critical here. If you forget that, you'll get byte strings instead of normal text strings.

### Commentary: Shouldn't we being using Pandas for this?

Data scientists are quick to point out that libraries like [Pandas](https://pandas.pydata.org/) already have a function for reading CSV files. This is true--and it works pretty well. However, this is not a course on learning Pandas. Reading files is a more general problem than the specifics of CSV files. The main reason we're working with a CSV file is that it's a familiar format to most coders and it's relatively easy to work with directly--illustrating many Python features in the process. So, by all means use Pandas when you go back to work. For the rest of this course however, we're going to stick with standard Python functionality.

[Contents](https://github.com/ashkan-abbasi66/python-comments/blob/master/prog/dabeaz/Notes/Contents.md) | [Previous (1.5 Lists)](https://github.com/ashkan-abbasi66/python-comments/blob/master/prog/dabeaz/Notes/01_Introduction/05_Lists.md) | [Next (1.7 Functions)](https://github.com/ashkan-abbasi66/python-comments/blob/master/prog/dabeaz/Notes/01_Introduction/07_Functions.md)

# 1.7 Functions

As your programs start to get larger, you'll want to get organized. This section briefly introduces functions and library modules. Error handling with exceptions is also introduced.

### Custom Functions

Use functions for code you want to reuse. Here is a function definition:

def sumcount(n):

'''

Returns the sum of the first n integers

'''

total = 0

while n > 0:

total += n

n -= 1

return total

To call a function.

a = sumcount(100)

A function is a series of statements that perform some task and return a result. The return keyword is needed to explicitly specify the return value of the function.

### Library Functions

Python comes with a large standard library. Library modules are accessed using import. For example:

import math

x = math.sqrt(10)

import urllib.request

u = urllib.request.urlopen('http://www.python.org/')

data = u.read()

We will cover libraries and modules in more detail later.

### Errors and exceptions

Functions report errors as exceptions. An exception causes a function to abort and may cause your entire program to stop if unhandled.

Try this in your python REPL.

>>> int('N/A')

Traceback (most recent call last):

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

ValueError: invalid literal for int() with base 10: 'N/A'

>>>

For debugging purposes, the message describes what happened, where the error occurred, and a traceback showing the other function calls that led to the failure.

### Catching and Handling Exceptions

Exceptions can be caught and handled.

To catch, use the try - except statement.

for line in f:

fields = line.split()

try:

shares = int(fields[1])

except ValueError:

print("Couldn't parse", line)

...

The name ValueError must match the kind of error you are trying to catch.

It is often difficult to know exactly what kinds of errors might occur in advance depending on the operation being performed. For better or for worse, exception handling often gets added after a program has unexpectedly crashed (i.e., "oh, we forgot to catch that error. We should handle that!").

### Raising Exceptions

To raise an exception, use the raise statement.

raise RuntimeError('What a kerfuffle')

This will cause the program to abort with an exception traceback. Unless caught by a try-except block.

% python3 foo.py

Traceback (most recent call last):

File "foo.py", line 21, in <module>

raise RuntimeError("What a kerfuffle")

RuntimeError: What a kerfuffle

## Exercises

### Exercise 1.29: Defining a function

Try defining a simple function:

>>> def greeting(name):

'Issues a greeting'

print('Hello', name)

>>> greeting('Guido')

Hello Guido

>>> greeting('Paula')

Hello Paula

>>>

If the first statement of a function is a string, it serves as documentation. Try typing a command such as help(greeting) to see it displayed.

### Exercise 1.30: Turning a script into a function

Take the code you wrote for the pcost.py program in [Exercise 1.27](https://github.com/ashkan-abbasi66/python-comments/blob/master/prog/dabeaz/Notes/01_Introduction/06_Files.md) and turn it into a function portfolio\_cost(filename). This function takes a filename as input, reads the portfolio data in that file, and returns the total cost of the portfolio as a float.

To use your function, change your program so that it looks something like this:

def portfolio\_cost(filename):

...

# Your code here

...

cost = portfolio\_cost('Data/portfolio.csv')

print('Total cost:', cost)

When you run your program, you should see the same output as before. After you’ve run your program, you can also call your function interactively by typing this:

bash $ python3 -i pcost.py

This will allow you to call your function from the interactive mode.

>>> portfolio\_cost('Data/portfolio.csv')

44671.15

>>>

Being able to experiment with your code interactively is useful for testing and debugging.

### Exercise 1.31: Error handling

What happens if you try your function on a file with some missing fields?

>>> portfolio\_cost('Data/missing.csv')

Traceback (most recent call last):

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

File "pcost.py", line 11, in portfolio\_cost

nshares = int(fields[1])

ValueError: invalid literal for int() with base 10: ''

>>>

At this point, you’re faced with a decision. To make the program work you can either sanitize the original input file by eliminating bad lines or you can modify your code to handle the bad lines in some manner.

Modify the pcost.py program to catch the exception, print a warning message, and continue processing the rest of the file.

### Exercise 1.32: Using a library function

Python comes with a large standard library of useful functions. One library that might be useful here is the csv module. You should use it whenever you have to work with CSV data files. Here is an example of how it works:

>>> import csv

>>> f = open('Data/portfolio.csv')

>>> rows = csv.reader(f)

>>> headers = next(rows)

>>> headers

['name', 'shares', 'price']

>>> for row in rows:

print(row)

['AA', '100', '32.20']

['IBM', '50', '91.10']

['CAT', '150', '83.44']

['MSFT', '200', '51.23']

['GE', '95', '40.37']

['MSFT', '50', '65.10']

['IBM', '100', '70.44']

>>> f.close()

>>>

One nice thing about the csv module is that it deals with a variety of low-level details such as quoting and proper comma splitting. In the above output, you’ll notice that it has stripped the double-quotes away from the names in the first column.

Modify your pcost.py program so that it uses the csv module for parsing and try running earlier examples.

### Exercise 1.33: Reading from the command line

In the pcost.py program, the name of the input file has been hardwired into the code:

# pcost.py

def portfolio\_cost(filename):

...

# Your code here

...

cost = portfolio\_cost('Data/portfolio.csv')

print('Total cost:', cost)

That’s fine for learning and testing, but in a real program you probably wouldn’t do that.

Instead, you might pass the name of the file in as an argument to a script. Try changing the bottom part of the program as follows:

# pcost.py

import sys

def portfolio\_cost(filename):

...

# Your code here

...

if len(sys.argv) == 2:

filename = sys.argv[1]

else:

filename = 'Data/portfolio.csv'

cost = portfolio\_cost(filename)

print('Total cost:', cost)

sys.argv is a list that contains passed arguments on the command line (if any).

To run your program, you’ll need to run Python from the terminal.

For example, from bash on Unix:

bash % python3 pcost.py Data/portfolio.csv

Total cost: 44671.15

bash %

[Contents](https://github.com/ashkan-abbasi66/python-comments/blob/master/prog/dabeaz/Notes/Contents.md) | [Previous (1.6 Files)](https://github.com/ashkan-abbasi66/python-comments/blob/master/prog/dabeaz/Notes/01_Introduction/06_Files.md) | [Next (2.0 Working with Data)](https://github.com/ashkan-abbasi66/python-comments/blob/master/prog/dabeaz/Notes/02_Working_with_data/00_Overview.md)

# 4. Classes and Objects

So far, our programs have only used built-in Python datatypes. In this section, we introduce the concept of classes and objects. You'll learn about the class statement that allows you to make new objects. We'll also introduce the concept of inheritance, a tool that is commonly use to build extensible programs. Finally, we'll look at a few other features of classes including special methods, dynamic attribute lookup, and defining new exceptions.

* [4.1 Introducing Classes](https://github.com/ashkan-abbasi66/python-comments/blob/master/prog/dabeaz/Notes/04_Classes_objects/01_Class.md)
* [4.2 Inheritance](https://github.com/ashkan-abbasi66/python-comments/blob/master/prog/dabeaz/Notes/04_Classes_objects/02_Inheritance.md)
* [4.3 Special Methods](https://github.com/ashkan-abbasi66/python-comments/blob/master/prog/dabeaz/Notes/04_Classes_objects/03_Special_methods.md)
* [4.4 Defining new Exception](https://github.com/ashkan-abbasi66/python-comments/blob/master/prog/dabeaz/Notes/04_Classes_objects/04_Defining_exceptions.md)

[Contents](https://github.com/ashkan-abbasi66/python-comments/blob/master/prog/dabeaz/Notes/Contents.md) | [Prev (3 Program Organization)](https://github.com/ashkan-abbasi66/python-comments/blob/master/prog/dabeaz/Notes/03_Program_organization/00_Overview.md) | [Next (5 Inner Workings of Python Objects)](https://github.com/ashkan-abbasi66/python-comments/blob/master/prog/dabeaz/Notes/05_Object_model/00_Overview.md)

# 4.1 Classes

This section introduces the class statement and the idea of creating new objects.

### Object Oriented (OO) programming

A Programming technique where code is organized as a collection of objects.

An object consists of:

* Data. Attributes
* Behavior. Methods which are functions applied to the object.

You have already been using some OO during this course.

For example, manipulating a list.

>>> nums = [1, 2, 3]

>>> nums.append(4) # Method

>>> nums.insert(1,10) # Method

>>> nums

[1, 10, 2, 3, 4] # Data

>>>

nums is an instance of a list.

Methods (append() and insert()) are attached to the instance (nums).

### The class statement

Use the class statement to define a new object.

class Player:

def \_\_init\_\_(self, x, y):

self.x = x

self.y = y

self.health = 100

def move(self, dx, dy):

self.x += dx

self.y += dy

def damage(self, pts):

self.health -= pts

In a nutshell, a class is a set of functions that carry out various operations on so-called instances.

### Instances

Instances are the actual objects that you manipulate in your program.

They are created by calling the class as a function.

>>> a = Player(2, 3)

>>> b = Player(10, 20)

>>>

a and b are instances of Player.

Emphasize: The class statement is just the definition (it does nothing by itself). Similar to a function definition.

### Instance Data

Each instance has its own local data.

>>> a.x

2

>>> b.x

10

This data is initialized by the \_\_init\_\_().

class Player:

def \_\_init\_\_(self, x, y):

# Any value stored on `self` is instance data

self.x = x

self.y = y

self.health = 100

There are no restrictions on the total number or type of attributes stored.

### Instance Methods

Instance methods are functions applied to instances of an object.

class Player:

...

# `move` is a method

def move(self, dx, dy):

self.x += dx

self.y += dy

The object itself is always passed as first argument.

>>> a.move(1, 2)

# matches `a` to `self`

# matches `1` to `dx`

# matches `2` to `dy`

def move(self, dx, dy):

By convention, the instance is called self. However, the actual name used is unimportant. The object is always passed as the first argument. It is merely Python programming style to call this argument self.

### Class Scoping

Classes do not define a scope of names.

class Player:

...

def move(self, dx, dy):

self.x += dx

self.y += dy

def left(self, amt):

move(-amt, 0) # NO. Calls a global `move` function

self.move(-amt, 0) # YES. Calls method `move` from above.

If you want to operate on an instance, you always refer to it explicitly (e.g., self).

## Exercises

Starting with this set of exercises, we start to make a series of changes to existing code from previous sections. It is critical that you have a working version of Exercise 3.18 to start. If you don't have that, please work from the solution code found in the Solutions/3\_18 directory. It's fine to copy it.

### Exercise 4.1: Objects as Data Structures

In section 2 and 3, we worked with data represented as tuples and dictionaries. For example, a holding of stock could be represented as a tuple like this:

s = ('GOOG',100,490.10)

or as a dictionary like this:

s = { 'name' : 'GOOG',

'shares' : 100,

'price' : 490.10

}

You can even write functions for manipulating such data. For example:

def cost(s):

return s['shares'] \* s['price']

However, as your program gets large, you might want to create a better sense of organization. Thus, another approach for representing data would be to define a class. Create a file called stock.py and define a class Stock that represents a single holding of stock. Have the instances of Stock have name, shares, and price attributes. For example:

>>> import stock

>>> a = stock.Stock('GOOG',100,490.10)

>>> a.name

'GOOG'

>>> a.shares

100

>>> a.price

490.1

>>>

Create a few more Stock objects and manipulate them. For example:

>>> b = stock.Stock('AAPL', 50, 122.34)

>>> c = stock.Stock('IBM', 75, 91.75)

>>> b.shares \* b.price

6117.0

>>> c.shares \* c.price

6881.25

>>> stocks = [a, b, c]

>>> stocks

[<stock.Stock object at 0x37d0b0>, <stock.Stock object at 0x37d110>, <stock.Stock object at 0x37d050>]

>>> for s in stocks:

print(f'{s.name:>10s} {s.shares:>10d} {s.price:>10.2f}')

... look at the output ...

>>>

One thing to emphasize here is that the class Stock acts like a factory for creating instances of objects. Basically, you call it as a function and it creates a new object for you. Also, it must be emphasized that each object is distinct---they each have their own data that is separate from other objects that have been created.

An object defined by a class is somewhat similar to a dictionary--just with somewhat different syntax. For example, instead of writing s['name'] or s['price'], you now write s.name and s.price.

### Exercise 4.2: Adding some Methods

With classes, you can attach functions to your objects. These are known as methods and are functions that operate on the data stored inside an object. Add a cost() and sell() method to your Stock object. They should work like this:

>>> import stock

>>> s = stock.Stock('GOOG', 100, 490.10)

>>> s.cost()

49010.0

>>> s.shares

100

>>> s.sell(25)

>>> s.shares

75

>>> s.cost()

36757.5

>>>

### Exercise 4.3: Creating a list of instances

Try these steps to make a list of Stock instances from a list of dictionaries. Then compute the total cost:

>>> import fileparse

>>> with open('Data/portfolio.csv') as lines:

... portdicts = fileparse.parse\_csv(lines, select=['name','shares','price'], types=[str,int,float])

...

>>> portfolio = [ stock.Stock(d['name'], d['shares'], d['price']) for d in portdicts]

>>> portfolio

[<stock.Stock object at 0x10c9e2128>, <stock.Stock object at 0x10c9e2048>, <stock.Stock object at 0x10c9e2080>,

<stock.Stock object at 0x10c9e25f8>, <stock.Stock object at 0x10c9e2630>, <stock.Stock object at 0x10ca6f748>,

<stock.Stock object at 0x10ca6f7b8>]

>>> sum([s.cost() for s in portfolio])

44671.15

>>>

### Exercise 4.4: Using your class

Modify the read\_portfolio() function in the report.py program so that it reads a portfolio into a list of Stock instances as just shown in Exercise 4.3. Once you have done that, fix all of the code in report.py and pcost.py so that it works with Stock instances instead of dictionaries.

Hint: You should not have to make major changes to the code. You will mainly be changing dictionary access such as s['shares'] into s.shares.

You should be able to run your functions the same as before:

>>> import pcost

>>> pcost.portfolio\_cost('Data/portfolio.csv')

44671.15

>>> import report

>>> report.portfolio\_report('Data/portfolio.csv', 'Data/prices.csv')

Name Shares Price Change

---------- ---------- ---------- ----------

AA 100 9.22 -22.98

IBM 50 106.28 15.18

CAT 150 35.46 -47.98

MSFT 200 20.89 -30.34

GE 95 13.48 -26.89

MSFT 50 20.89 -44.21

IBM 100 106.28 35.84

>>>

[Contents](https://github.com/ashkan-abbasi66/python-comments/blob/master/prog/dabeaz/Notes/Contents.md) | [Previous (3.6 Design discussion)](https://github.com/ashkan-abbasi66/python-comments/blob/master/prog/dabeaz/Notes/03_Program_organization/06_Design_discussion.md) | [Next (4.2 Inheritance)](https://github.com/ashkan-abbasi66/python-comments/blob/master/prog/dabeaz/Notes/04_Classes_objects/02_Inheritance.md)