Welcome to day 2 of the Learn Python challenge! If you missed day 1, you can find the notebook [here](https://www.kaggle.com/colinmorris/learn-python-challenge-day-1/notebook).

Today we'll be talking about functions: calling them, defining them, and looking them up using Python's built-in documentation.

In some languages, functions must be defined to always take a specific number of arguments, each having a particular type. Python functions are allowed much more flexibility. The print function is a good example of this:

In [1]:

print("The print function takes an input and prints it to the screen.")

print("Each call to print starts on a new line.")

print("You'll often call print with strings, but you can pass any kind of value. For example, a number:")

print(2 + 2)

print("If print is called with multiple arguments...", "it joins them",

"(with spaces in between)", "before printing.")

print('But', 'this', 'is', 'configurable', sep='!...')

print()

print("^^^ print can also be called with no arguments to print a blank line.")

The print function takes an input and prints it to the screen.

Each call to print starts on a new line.

You'll often call print with strings, but you can pass any kind of value. For example, a number:

4

If print is called with multiple arguments... it joins them (with spaces in between) before printing.

But!...this!...is!...configurable

^^^ print can also be called with no arguments to print a blank line.

"What does this function do again?"

I showed the abs function in the previous lesson, but what if you've forgotten what it does?

The help() function is possibly the most important Python function you can learn. If you can remember how to use help(), you hold the key to understanding just about any other function in Python.

In [2]:

help(abs)

Help on built-in function abs in module builtins:

abs(x, /)

Return the absolute value of the argument.

When applied to a function, help() displays...

* the header of that function abs(x, /). In this case, this tells us that abs() takes a single argument x. (The forward slash isn't important, but if you're curious, you can read about it [here](https://stackoverflow.com/questions/24735311/python-what-does-the-slash-mean-in-the-output-of-helprange))
* A brief English description of what the function does.

**Common pitfall:** when you're looking up a function, remember to pass in the name of the function itself, and not the result of calling that function.

What happens if we invoke help on a *call* to the function abs()? Unhide the output of the cell below to see.

Hide

In [3]:

help(abs(-2))

Help on int object:

class int(object)

| int(x=0) -> integer

| int(x, base=10) -> integer

|

| Convert a number or string to an integer, or return 0 if no arguments

| are given. If x is a number, return x.\_\_int\_\_(). For floating point

| numbers, this truncates towards zero.

|

| If x is not a number or if base is given, then x must be a string,

| bytes, or bytearray instance representing an integer literal in the

| given base. The literal can be preceded by '+' or '-' and be surrounded

| by whitespace. The base defaults to 10. Valid bases are 0 and 2-36.

| Base 0 means to interpret the base from the string as an integer literal.

| >>> int('0b100', base=0)

| 4

|

| Methods defined here:

|

| \_\_abs\_\_(self, /)

| abs(self)

|

| \_\_add\_\_(self, value, /)

| Return self+value.

|

| \_\_and\_\_(self, value, /)

| Return self&value.

|

| \_\_bool\_\_(self, /)

| self != 0

|

| \_\_ceil\_\_(...)

| Ceiling of an Integral returns itself.

|

| \_\_divmod\_\_(self, value, /)

| Return divmod(self, value).

|

| \_\_eq\_\_(self, value, /)

| Return self==value.

|

| \_\_float\_\_(self, /)

| float(self)

|

| \_\_floor\_\_(...)

| Flooring an Integral returns itself.

|

| \_\_floordiv\_\_(self, value, /)

| Return self//value.

|

| \_\_format\_\_(...)

| default object formatter

|

| \_\_ge\_\_(self, value, /)

| Return self>=value.

|

| \_\_getattribute\_\_(self, name, /)

| Return getattr(self, name).

|

| \_\_getnewargs\_\_(...)

|

| \_\_gt\_\_(self, value, /)

| Return self>value.

|

| \_\_hash\_\_(self, /)

| Return hash(self).

|

| \_\_index\_\_(self, /)

| Return self converted to an integer, if self is suitable for use as an index into a list.

|

| \_\_int\_\_(self, /)

| int(self)

|

| \_\_invert\_\_(self, /)

| ~self

|

| \_\_le\_\_(self, value, /)

| Return self<=value.

|

| \_\_lshift\_\_(self, value, /)

| Return self<<value.

|

| \_\_lt\_\_(self, value, /)

| Return self<value.

|

| \_\_mod\_\_(self, value, /)

| Return self%value.

|

| \_\_mul\_\_(self, value, /)

| Return self\*value.

|

| \_\_ne\_\_(self, value, /)

| Return self!=value.

|

| \_\_neg\_\_(self, /)

| -self

|

| \_\_new\_\_(\*args, \*\*kwargs) from builtins.type

| Create and return a new object. See help(type) for accurate signature.

|

| \_\_or\_\_(self, value, /)

| Return self|value.

|

| \_\_pos\_\_(self, /)

| +self

|

| \_\_pow\_\_(self, value, mod=None, /)

| Return pow(self, value, mod).

|

| \_\_radd\_\_(self, value, /)

| Return value+self.

|

| \_\_rand\_\_(self, value, /)

| Return value&self.

|

| \_\_rdivmod\_\_(self, value, /)

| Return divmod(value, self).

|

| \_\_repr\_\_(self, /)

| Return repr(self).

|

| \_\_rfloordiv\_\_(self, value, /)

| Return value//self.

|

| \_\_rlshift\_\_(self, value, /)

| Return value<<self.

|

| \_\_rmod\_\_(self, value, /)

| Return value%self.

|

| \_\_rmul\_\_(self, value, /)

| Return value\*self.

|

| \_\_ror\_\_(self, value, /)

| Return value|self.

|

| \_\_round\_\_(...)

| Rounding an Integral returns itself.

| Rounding with an ndigits argument also returns an integer.

|

| \_\_rpow\_\_(self, value, mod=None, /)

| Return pow(value, self, mod).

|

| \_\_rrshift\_\_(self, value, /)

| Return value>>self.

|

| \_\_rshift\_\_(self, value, /)

| Return self>>value.

|

| \_\_rsub\_\_(self, value, /)

| Return value-self.

|

| \_\_rtruediv\_\_(self, value, /)

| Return value/self.

|

| \_\_rxor\_\_(self, value, /)

| Return value^self.

|

| \_\_sizeof\_\_(...)

| Returns size in memory, in bytes

|

| \_\_str\_\_(self, /)

| Return str(self).

|

| \_\_sub\_\_(self, value, /)

| Return self-value.

|

| \_\_truediv\_\_(self, value, /)

| Return self/value.

|

| \_\_trunc\_\_(...)

| Truncating an Integral returns itself.

|

| \_\_xor\_\_(self, value, /)

| Return self^value.

|

| bit\_length(...)

| int.bit\_length() -> int

|

| Number of bits necessary to represent self in binary.

| >>> bin(37)

| '0b100101'

| >>> (37).bit\_length()

| 6

|

| conjugate(...)

| Returns self, the complex conjugate of any int.

|

| from\_bytes(...) from builtins.type

| int.from\_bytes(bytes, byteorder, \*, signed=False) -> int

|

| Return the integer represented by the given array of bytes.

|

| The bytes argument must be a bytes-like object (e.g. bytes or bytearray).

|

| The byteorder argument determines the byte order used to represent the

| integer. If byteorder is 'big', the most significant byte is at the

| beginning of the byte array. If byteorder is 'little', the most

| significant byte is at the end of the byte array. To request the native

| byte order of the host system, use `sys.byteorder' as the byte order value.

|

| The signed keyword-only argument indicates whether two's complement is

| used to represent the integer.

|

| to\_bytes(...)

| int.to\_bytes(length, byteorder, \*, signed=False) -> bytes

|

| Return an array of bytes representing an integer.

|

| The integer is represented using length bytes. An OverflowError is

| raised if the integer is not representable with the given number of

| bytes.

|

| The byteorder argument determines the byte order used to represent the

| integer. If byteorder is 'big', the most significant byte is at the

| beginning of the byte array. If byteorder is 'little', the most

| significant byte is at the end of the byte array. To request the native

| byte order of the host system, use `sys.byteorder' as the byte order value.

|

| The signed keyword-only argument determines whether two's complement is

| used to represent the integer. If signed is False and a negative integer

| is given, an OverflowError is raised.

|

| ----------------------------------------------------------------------

| Data descriptors defined here:

|

| denominator

| the denominator of a rational number in lowest terms

|

| imag

| the imaginary part of a complex number

|

| numerator

| the numerator of a rational number in lowest terms

|

| real

| the real part of a complex number

Python evaluates an expression like this from the inside out. First it calculates the value of abs(-2), then it provides help on whatever the value of that expression is.

(And it turns out to have a lot to say about integers! In Python, even something as simple-seeming as an integer is actually an object with a fair amount of internal complexity. After we talk later about objects, methods, and attributes in Python, the voluminous help output above will make more sense.)

abs is a very simple function with a short docstring. help shines even more when dealing with more complex, configurable functions like print:

In [4]:

help(print)

Help on built-in function print in module builtins:

print(...)

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

Prints the values to a stream, or to sys.stdout by default.

Optional keyword arguments:

file: a file-like object (stream); defaults to the current sys.stdout.

sep: string inserted between values, default a space.

end: string appended after the last value, default a newline.

flush: whether to forcibly flush the stream.

Some of this might look inscrutable for now (what's sys.stdout?), but this docstring does shed some light on that sep parameter we used in one of our print examples at the beginning.

Defining functions

Builtin functions are great, but we can only get so far with them before we need to start defining our own functions. Below is a simple example.

In [5]:

def least\_difference(a, b, c):

diff1 = abs(a - b)

diff2 = abs(b - c)

diff3 = abs(a - c)

return min(diff1, diff2, diff3)

This creates a function called least\_difference, which takes three arguments, a, b, and c.

Functions start with a header introduced by the def keyword. The indented block of code following the : is run when the function is called.

return is another keyword uniquely associated with functions. When Python encounters a return statement, it exits the function immediately, and passes the value on the right hand side to the calling context.

Is it clear what least\_difference() does from the source code? If we're not sure, we can always try it out on a few examples:

In [6]:

print(

least\_difference(1, 10, 100),

least\_difference(1, 10, 10),

least\_difference(5, 6, 7), *# Python allows trailing commas in argument lists. How nice is that?*

)

9 0 1

Or maybe the help() function can tell us something about it.

In [7]:

help(least\_difference)

Help on function least\_difference in module \_\_main\_\_:

least\_difference(a, b, c)

Unsurprisingly, Python isn't smart enough to read my code and turn it into a nice English description. However, when I write a function, I can provide a description in what's called the **docstring**.

Docstrings

In [8]:

def least\_difference(a, b, c):

*"""Return the smallest difference between any two numbers*

*among a, b and c.*

*>>> least\_difference(1, 5, -5)*

*4*

*"""*

diff1 = abs(a - b)

diff2 = abs(b - c)

diff3 = abs(a - c)

return min(diff1, diff2, diff3)

The docstring is a triple-quoted string (which may span multiple lines) that comes immediately after the header of a function. When we call help() on a function, it shows the docstring.

In [9]:

help(least\_difference)

Help on function least\_difference in module \_\_main\_\_:

least\_difference(a, b, c)

Return the smallest difference between any two numbers

among a, b and c.

>>> least\_difference(1, 5, -5)

4

**Aside: example calls** The last two lines of the docstring are an example function call and result. (The >>> is a reference to the command prompt used in Python interactive shells.) Python doesn't run the example call - it's just there for the benefit of the reader. The convention of including 1 or more example calls in a function's docstring is far from universally observed, but it can be very effective at helping someone understand your function. For a real-world example of, see [this docstring for the numpy function np.eye](https://github.com/numpy/numpy/blob/v1.14.2/numpy/lib/twodim_base.py#L140-L194).

Docstrings are a nice way to document your code for others - or even for yourself. (How many times have you come back to some code you were working on a day ago and wondered "what was I thinking?")

Functions that don't return

What would happen if we didn't include the return keyword in our function?

In [10]:

def least\_difference(a, b, c):

*"""Return the smallest difference between any two numbers*

*among a, b and c.*

*"""*

diff1 = abs(a - b)

diff2 = abs(b - c)

diff3 = abs(a - c)

min(diff1, diff2, diff3)

print(

least\_difference(1, 10, 100),

least\_difference(1, 10, 10),

least\_difference(5, 6, 7),

)

None None None

Python allows us to define such functions. The result of calling them is the special value None. (This is similar to the concept of "null" in other languages.)

Without a return statement, least\_difference is completely pointless, but a function with side effects may do something useful without returning anything. We've already seen two examples of this: print() and help() don't return anything. We only call them for their side effects (putting some text on the screen). Other examples of useful side effects include writing to a file, or modifying an input.

In [11]:

mystery = print()

print(mystery)

None

Default arguments

When we called help(print), we saw that the print function has several optional arguments. For example, we can specify a value for sep to put some special string in between our printed arguments:

In [12]:

print(1, 2, 3, sep=' < ')

1 < 2 < 3

But if we don't specify a value, sep is treated as having a default value of ' ' (a single space).

In [13]:

print(1, 2, 3)

1 2 3

Adding optional arguments with default values to the functions we define turns out to be pretty easy:

In [14]:

def greet(who="Colin"):

print("Hello,", who)

greet()

greet(who="Kaggle")

*# (In this case, we don't need to specify the name of the argument, because it's unambiguous.)*

greet("world")

Hello, Colin

Hello, Kaggle

Hello, world

Functions are objects too

In [15]:

def f(n):

return n \* 2

x = 12.5

The syntax for creating them may be different, but f and x in the code above aren't so fundamentally different. They're each variables that refer to objects. x refers to an object of type float, and f refers to an object of type... well, let's ask Python:

In [16]:

print(

type(x),

type(f), sep='**\n**'

)

<class 'float'>

<class 'function'>

We can even ask Python to print f out:

In [17]:

print(x)

print(f)

12.5

<function f at 0x7f985ddaa950>

...though what it shows isn't super useful.

Notice that the code cells above have examples of functions (type, and print) taking *another function* as input. This opens up some interesting possibilities - we can *call* the function we receive as an argument.

In [18]:

def call(fn, arg):

*"""Call fn on arg"""*

return fn(arg)

def squared\_call(fn, arg):

*"""Call fn on the result of calling fn on arg"""*

return fn(fn(arg))

print(

call(f, 1),

squared\_call(f, 1),

sep='**\n**', *# '\n' is the newline character - it starts a new line*

)

2

4

You probably won't often define [higher order functions](https://en.wikipedia.org/wiki/Higher-order_function) like this yourself, but there are some existing ones (built in to Python and in libraries like pandas or numpy) that you might find useful to call. For example, max.

By default, max returns the largest of its arguments. But if we pass in a function using the optional key argument, it returns the argument xthat maximizes key(x) (aka the 'argmax').

In [19]:

def mod\_5(x):

*"""Return the remainder of x after dividing by 5"""*

return x % 5

print(

'Which number is biggest?',

max(100, 51, 14),

'Which number is the biggest modulo 5?',

max(100, 51, 14, key=mod\_5),

sep='**\n**',

)

Which number is biggest?

100

Which number is the biggest modulo 5?

14

Lambda functions

If you're writing a short throwaway function whose body fits into a single line (like mod\_5 above), Python's lambda syntax is conveniently compact.

In [20]:

mod\_5 = lambda x: x % 5

*# Note that we don't use the "return" keyword above (it's implicit)*

*# (The line below would produce a SyntaxError)*

*#mod\_5 = lambda x: return x % 5*

print('101 mod 5 =', mod\_5(101))

101 mod 5 = 1

In [21]:

*# Lambdas can take multiple comma-separated arguments*

abs\_diff = lambda a, b: abs(a-b)

print("Absolute difference of 5 and 7 is", abs\_diff(5, 7))

Absolute difference of 5 and 7 is 2

In [22]:

*# Or no arguments*

always\_32 = lambda: 32

always\_32()

Out[22]:

32

With judicious use of lambdas, you can occasionally solve complex problems in a single line.

In [23]:

*# Preview of lists and strings. (We'll go in depth into both soon)*

*# - len: return the length of a sequence (such as a string or list)*

*# - sorted: return a sorted version of the given sequence (optional key*

*# function works similarly to max and min)*

*# - s.lower() : return a lowercase version of string s*

names = ['jacques', 'Ty', 'Mia', 'pui-wa']

print("Longest name is:", max(names, key=lambda name: len(name))) *# or just key=len*

print("Names sorted case insensitive:", sorted(names, key=lambda name: name.lower()))

Longest name is: jacques

Names sorted case insensitive: ['jacques', 'Mia', 'pui-wa', 'Ty']

Your Turn

To get started on the day 2 exercises, head over to [this notebook](https://www.kaggle.com/colinmorris/learn-python-challenge-day-2-exercises) and click the "Fork Notebook" button.

# Exercise 2

These exercises accompany the tutorial on [functions and getting help](https://www.kaggle.com/colinmorris/functions-and-getting-help-daily).

As before, don't forget to run the setup code below before jumping into question 1.

Hide

Hide

In [1]:

*# SETUP. You don't need to worry for now about what this code does or how it works.*

from learntools.core import binder; binder.bind(globals())

from learntools.python.ex2 import \*

print('Setup complete.')

Setup complete.

# **Exercises**

## 1.

Complete the body of the following function according to its docstring.

HINT: Python has a builtin function round

In [2]:

def round\_to\_two\_places(num):

*"""Return the given number rounded to two decimal places.*

*>>> round\_to\_two\_places(3.14159)*

*3.14*

*"""*

*# Replace this body with your own code.*

*# ("pass" is a keyword that does literally nothing. We used it as a placeholder*

*# because after we begin a code block, Python requires at least one line of code)*

pass

q1.check()

Check: When you've updated the starter code, check()will tell you whether your code is correct.

In [3]:

*# Uncomment the following for a hint*

*#q1.hint()*

*# Or uncomment the following to peek at the solution*

*#q1.solution()*

## 2.

The help for round says that ndigits (the second argument) may be negative. What do you think will happen when it is? Try some examples in the following cell?

Can you think of a case where this would be useful?

In [4]:

*# Put your test code here*

In [5]:

*#q2.solution()*

## 3.

In a previous programming problem, the candy-sharing friends Alice, Bob and Carol tried to split candies evenly. For the sake of their friendship, any candies left over would be smashed. For example, if they collectively bring home 91 candies, they'll take 30 each and smash 1.

Below is a simple function that will calculate the number of candies to smash for any number of total candies.

Modify it so that it optionally takes a second argument representing the number of friends the candies are being split between. If no second argument is provided, it should assume 3 friends, as before.

Update the docstring to reflect this new behaviour.

In [6]:

def to\_smash(total\_candies):

*"""Return the number of leftover candies that must be smashed after distributing*

*the given number of candies evenly between 3 friends.*

*>>> to\_smash(91)*

*1*

*"""*

return total\_candies % 3

q3.check()

Incorrect: to\_smash should be callable with two arguments (e.g. to\_smash(10, 2)

In [7]:

*#q3.hint()*

In [8]:

*#q3.solution()*

## 4.

It may not be fun, but reading and understanding error messages will be an important part of your Python career.

Each code cell below contains some commented-out buggy code. For each cell...

1. Read the code and predict what you think will happen when it's run.
2. Then uncomment the code and run it to see what happens. (**Tip**: In the kernel editor, you can highlight several lines and press ctrl+/ to toggle commenting.)
3. Fix the code (so that it accomplishes its intended purpose without throwing an exception)

In [9]:

*# ruound\_to\_two\_places(9.9999)*

In [10]:

*# x = -10*

*# y = 5*

*# # Which of the two variables above has the smallest absolute value?*

*# smallest\_abs = min(abs(x, y))*

In [11]:

*# def f(x):*

*# y = abs(x)*

*# return y*

*# print(f(5))*

## 5. 🌶️

For this question, we'll be using two functions imported from Python's time module.

The [time](https://docs.python.org/3/library/time.html#time.time) function returns the number of seconds that have passed since the Epoch (aka [Unix time](https://en.wikipedia.org/wiki/Unix_time)).

Try it out below. Each time you run it, you should get a slightly larger number.

In [12]:

*# Importing the function 'time' from the module of the same name.*

*# (We'll discuss imports in more depth later)*

from time import time

t = time()

print(t, "seconds since the Epoch")

1555540255.0669878 seconds since the Epoch

We'll also be using a function called [sleep](https://docs.python.org/3/library/time.html#time.sleep), which makes us wait some number of seconds while it does nothing particular. (Sounds useful, right?)

You can see it in action by running the cell below:

In [13]:

from time import sleep

duration = 5

print("Getting sleepy. See you in", duration, "seconds")

sleep(duration)

print("I'm back. What did I miss?")

Getting sleepy. See you in 5 seconds

I'm back. What did I miss?

With the help of these functions, complete the function time\_callbelow according to its docstring.

In [14]:

def time\_call(fn, arg):

*"""Return the amount of time the given function takes (in seconds) when called with the given argument.*

*"""*

pass

How would you verify that time\_call is working correctly? Think about it, and then check the answer with the solution function below`.

In [15]:

*#q5.hint()*

*#q5.solution()*

## 6. 🌶️

Note: this question depends on a working solution to the previous question.

Complete the function below according to its docstring.

In [16]:

def slowest\_call(fn, arg1, arg2, arg3):

*"""Return the amount of time taken by the slowest of the following function*

*calls: fn(arg1), fn(arg2), fn(arg3)*

*"""*

pass

In [17]:

*#q6.hint()*

In [18]:

*#q6.solution()*

**You'll get another email tomorrow so you can keep learning. See you then.**

# Day 3

# **Booleans**

Python has a type of variable called bool. It has two possible values: Trueand False.

In [1]:

x = True

print(x)

print(type(x))

True

<class 'bool'>

Rather than putting True or False directly in our code, we usually get boolean values from **boolean operators**. These are operators that answer yes/no questions. We'll go through some of these operators below.

## Comparison Operations

| Operation | Description |  | Operation | Description |
| --- | --- | --- | --- | --- |
| a == b | a equal to b |  | a != b | a not equal to b |
| a < b | a less than b |  | a > b | a greater than b |
| a <= b | a less than or equal to b |  | a >= b | a greater than or equal to b |

In [2]:

def can\_run\_for\_president(age):

*"""Can someone of the given age run for president in the US?"""*

*# The US Constitution says you must be at least 35 years old*

return age >= 35

print("Can a 19-year-old run for president?", can\_run\_for\_president(19))

print("Can a 45-year-old run for president?", can\_run\_for\_president(45))

Can a 19-year-old run for president? False

Can a 45-year-old run for president? True

Comparisons frequently work like you'd hope

In [3]:

3.0 == 3

Out[3]:

True

But sometimes they can be tricky

In [4]:

'3' == 3

Out[4]:

False

Comparison operators can be combined with the arithmetic operators we've already seen to express a virtually limitless range of mathematical tests. For example, we can check if a number is odd by checking that the modulus with 2 returns 1:

In [5]:

def is\_odd(n):

return (n % 2) == 1

print("Is 100 odd?", is\_odd(100))

print("Is -1 odd?", is\_odd(-1))

Is 100 odd? False

Is -1 odd? True

Remember to use == instead of = when making comparisons. If you write n == 2 you are asking about the value of n. When you write n = 2 you are changing the value of n.

## Combining Boolean Values

You can combine boolean values using the standard concepts of "and", "or", and "not". In fact, the words to do this are: and, or, and not.

With these, we can make our can\_run\_for\_president function more accurate.

In [6]:

def can\_run\_for\_president(age, is\_natural\_born\_citizen):

*"""Can someone of the given age and citizenship status run for president in the US?"""*

*# The US Constitution says you must be a natural born citizen \*and\* at least 35 years old*

return is\_natural\_born\_citizen **and** (age >= 35)

print(can\_run\_for\_president(19, True))

print(can\_run\_for\_president(55, False))

print(can\_run\_for\_president(55, True))

False

False

True

Quick, can you guess the value of this expression?

Output

In [7]:

True **or** True **and** False

(Click the "output" button to see the answer)

To answer this, you'd need to figure out the order of operations.

For example, and is evaluated before or. That's why the first expression above is True. If we evaluated it from left to right, we would have calculated True or True first (which is True), and then taken the and of that result with False, giving a final value of False.

You could try to [memorize the order of precedence](https://docs.python.org/3/reference/expressions.html#operator-precedence), but a safer bet is to just use liberal parentheses. Not only does this help prevent bugs, it makes your intentions clearer to anyone who reads your code.

For example, consider the following expression:

prepared\_for\_weather = have\_umbrella **or** rain\_level < 5 **and** have\_hood **or** **not** rain\_level > 0 **and** is\_workday

I'm trying to say that I'm safe from today's weather....

* if I have an umbrella...
* or if the rain isn't too heavy and I have a hood...
* otherwise, I'm still fine unless it's raining and it's a workday

But not only is my Python code hard to read, it has a bug. We can address both problems by adding some parentheses:

prepared\_for\_weather = have\_umbrella **or** (rain\_level < 5 **and** have\_hood) **or** **not** (rain\_level > 0 **and** is\_workday)

You can add even more parentheses if you think it helps readability:

prepared\_for\_weather = have\_umbrella **or** ((rain\_level < 5) **and** have\_hood) **or** (**not** (rain\_level > 0 **and** is\_workday))

We can also split it over multiple lines to emphasize the 3-part structure described above:

prepared\_for\_weather = (

have\_umbrella

**or** ((rain\_level < 5) **and** have\_hood)

**or** (**not** (rain\_level > 0 **and** is\_workday))

)

# **Conditionals**

Booleans are most useful when combined with conditional statements, using the keywords if, elif, and else.

Conditional statements, often referred to as if-then statements, let you control what pieces of code are run based on the value of some Boolean condition. Here's an example:

In [8]:

def inspect(x):

if x == 0:

print(x, "is zero")

elif x > 0:

print(x, "is positive")

elif x < 0:

print(x, "is negative")

else:

print(x, "is unlike anything I've ever seen...")

inspect(0)

inspect(-15)

0 is zero

-15 is negative

The if and else keywords are often used in other languages; its more unique keyword is elif, a contraction of "else if". In these conditional clauses, elif and else blocks are optional; additionally, you can include as many elif statements as you would like.

Note especially the use of colons (:) and whitespace to denote separate blocks of code. This is similar to what happens when we define a function - the function header ends with :, and the following line is indented with 4 spaces. All subsequent indented lines belong to the body of the function, until we encounter an unindented line, ending the function definition.

In [9]:

def f(x):

if x > 0:

print("Only printed when x is positive; x =", x)

print("Also only printed when x is positive; x =", x)

print("Always printed, regardless of x's value; x =", x)

f(1)

f(0)

Only printed when x is positive; x = 1

Also only printed when x is positive; x = 1

Always printed, regardless of x's value; x = 1

Always printed, regardless of x's value; x = 0

## Boolean conversion

We've seen int(), which turns things into ints, and float(), which turns things into floats, so you might not be surprised to hear that Python has a bool() function which turns things into bools.

In [10]:

print(bool(1)) *# all numbers are treated as true, except 0*

print(bool(0))

print(bool("asf")) *# all strings are treated as true, except the empty string ""*

print(bool(""))

*# Generally empty sequences (strings, lists, and other types we've yet to see like lists and tuples)*

*# are "falsey" and the rest are "truthy"*

True

False

True

False

We can use non-boolean objects in if conditions and other places where a boolean would be expected. Python will implicitly treat them as their corresponding boolean value:

In [11]:

if 0:

print(0)

elif "spam":

print("spam")

spam

# **Your Turn**

Try the [hands-on exercise](https://www.kaggle.com/kernels/fork/958279) with booleans and conditionals

# DAY 4

Welcome to day 4 of the Python Challenge! If you missed any of the previous days, here are the links:

* [Day 1 (syntax, variable assignment, numbers)](https://www.kaggle.com/colinmorris/learn-python-challenge-day-1)
* [Day 2 (functions and getting help)](https://www.kaggle.com/colinmorris/learn-python-challenge-day-2)
* [Day 3 (booleans and conditionals)](https://www.kaggle.com/colinmorris/learn-python-challenge-day-3)

Today we'll be talking about **lists**.

Lists in Python represent ordered sequences of values. They can be defined with comma-separated values between square brackets. For example, here is a list of the first few prime numbers:

In [1]:

primes = [2, 3, 5, 7]

We can put other types of things in lists:

In [2]:

planets = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']

Including other lists:

In [3]:

hands = [

['J', 'Q', 'K'],

['2', '2', '2'],

['6', 'A', 'K'], *# (Comma after the last element is optional)*

]

*# (I could also have written this on one line, but it can get hard to read)*

hands = [['J', 'Q', 'K'], ['2', '2', '2'], ['6', 'A', 'K']]

A list can contain a mix of different types:

In [4]:

my\_favourite\_things = [32, 'raindrops on roses', help]

*# (Yes, Python's help function is \*definitely\* one of my favourite things)*

## Indexing

We can access individual list elements using Python's square bracket indexing syntax.

Which planet is closest to the sun? Python uses zero-based indexing, so the first element has index 0.

In [5]:

planets[0]

Out[5]:

'Mercury'

What's the next closest planet?

In [6]:

planets[1]

Out[6]:

'Venus'

Which planet is furthest from the sun?

Elements at the end of the list can be accessed with negative numbers, starting from -1:

In [7]:

planets[-1]

Out[7]:

'Neptune'

In [8]:

planets[-2]

Out[8]:

'Uranus'

## Slicing

What are the first three planets? We can answer this question using slicing:

In [9]:

planets[0:3]

Out[9]:

['Mercury', 'Venus', 'Earth']

planets[0:3] is our way of asking Python for the elements of planetsstarting from index 0 and continuing up to but not including index 3.

The starting and ending indices are both optional. If I leave out the start index, it's assumed to be 0. So I could rewrite the expression above as:

In [10]:

planets[:3]

Out[10]:

['Mercury', 'Venus', 'Earth']

If I leave out the end index, it's assumed to be the length of the list.

In [11]:

planets[3:]

Out[11]:

['Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']

i.e. the expression above means "give me all the planets from index 3 onward".

We can also use negative indices when slicing:

In [12]:

*# All the planets except the first and last*

planets[1:-1]

Out[12]:

['Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus']

In [13]:

*# The last 3 planets*

planets[-3:]

Out[13]:

['Saturn', 'Uranus', 'Neptune']

## Mutating lists

Lists are mutable, meaning they can be modified "in place".

One way to modify a list is to assign to an index or slice expression.

For example, let's say we want to rename Mars:

In [14]:

planets[3] = 'Malacandra'

planets

Out[14]:

['Mercury',

'Venus',

'Earth',

'Malacandra',

'Jupiter',

'Saturn',

'Uranus',

'Neptune']

Hm, that's quite a mouthful. Let's compensate by shortening the names of the first 3 planets.

In [15]:

planets[:3] = ['Mur', 'Vee', 'Ur']

print(planets)

*# (Okay, that was rather silly. Let's give them back their old names)*

planets[:4] = ['Mercury', 'Venus', 'Earth', 'Mars',]

['Mur', 'Vee', 'Ur', 'Malacandra', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']

## List functions

Python has several useful builtin functions for working with lists.

len gives the length of a list:

In [16]:

*# How many planets are there?*

len(planets)

Out[16]:

8

sorted returns a sorted version of a list:

In [17]:

*# The planets sorted in alphabetical order*

sorted(planets)

Out[17]:

['Earth', 'Jupiter', 'Mars', 'Mercury', 'Neptune', 'Saturn', 'Uranus', 'Venus']

sum does what you might expect:

In [18]:

primes = [2, 3, 5, 7]

sum(primes)

Out[18]:

17

We've previously used the min and max to get the minimum or maximum of several arguments. But we can also pass in a single list argument.

In [19]:

max(primes)

Out[19]:

7

## Interlude: objects

I've been throwing around the term 'object' a lot so far - you may even remember me claiming that everything in Python is an object. But what does that actually mean?

In short, objects carry some stuff around with them. We can access that stuff using Python's dot syntax.

For example, numbers in Python carry around an associated variable called imag representing their imaginary part. (You'll probably never need to use this unless you're doing some very weird math.)

In [20]:

x = 12

*# x is a real number, so its imaginary part is 0.*

print(x.imag)

*# Here's how to make a complex number, in case you've ever been curious:*

c = 12 + 3j

print(c.imag)

0

3.0

The things an object carries around can also include functions. A function attached to an object is called a **method**. (Non-function things attached to an object, such as imag, are called attributes).

For example, numbers have a method called bit\_length. Again, we access it using dot syntax:

In [21]:

x.bit\_length

Out[21]:

<function int.bit\_length>

To actually call it, we add parentheses:

In [22]:

x.bit\_length()

Out[22]:

4

**Aside:** You've actually been calling methods for the last few days if you've been doing the exercises. In the exercise notebooks q1, q2, q3, etc. are all objects which have methods called check, hint, and solution.

In the same way that we can pass functions to the help function (e.g. help(max)), we can also pass in methods:

In [23]:

help(x.bit\_length)

Help on built-in function bit\_length:

bit\_length(...) method of builtins.int instance

int.bit\_length() -> int

Number of bits necessary to represent self in binary.

>>> bin(37)

'0b100101'

>>> (37).bit\_length()

6

The examples above were utterly obscure. None of the types of objects we've looked at so far (numbers, functions, booleans) have attributes or methods you're likely ever to use.

But it turns out that lists have several methods which you'll use all the time.

## List methods

list.append modifies a list by adding an item to the end:

In [24]:

*# Pluto is a planet darn it!*

planets.append('Pluto')

Why does the cell above have no output? Let's check the documentation by calling help(planets.append).

**Aside:** append is a method carried around by all objects of type list, not just planets, so we also could have called help(list.append). However, if we try to call help(append), Python will complain that no variable exists called "append". The "append" name only exists within lists - it doesn't exist as a standalone name like builtin functions such as max or len.

In [25]:

help(planets.append)

Help on built-in function append:

append(...) method of builtins.list instance

L.append(object) -> None -- append object to end

The -> None part is telling us that list.append doesn't return anything. But if we check the value of planets, we can see that the method call modified the value of planets:

In [26]:

planets

Out[26]:

['Mercury',

'Venus',

'Earth',

'Mars',

'Jupiter',

'Saturn',

'Uranus',

'Neptune',

'Pluto']

list.pop removes and returns the last element of a list:

In [27]:

planets.pop()

Out[27]:

'Pluto'

In [28]:

planets

Out[28]:

['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']

### Searching lists

Where does Earth fall in the order of planets? We can get its index using the list.index method.

In [29]:

planets.index('Earth')

Out[29]:

2

It comes third (i.e. at index 2 - 0 indexing!).

At what index does Pluto occur?

In [30]:

planets.index('Pluto')

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

ValueError Traceback (most recent call last)

<ipython-input-30-3a8584cba91a> in <module>()

----> 1 planets.index('Pluto')

ValueError: 'Pluto' is not in list

Oh, that's right...

To avoid unpleasant surprises like this, we can use the in operator to determine whether a list contains a particular value:

In [31]:

*# Is Earth a planet?*

"Earth" **in** planets

Out[31]:

True

In [32]:

*# Is Calbefraques a planet?*

"Calbefraques" **in** planets

Out[32]:

False

There are a few more interesting list methods we haven't covered. If you want to learn about all the methods and attributes attached to a particular object, we can call help() on the object itself. For example, help(planets) will tell us about all the list methods:

Output

In [33]:

help(planets)

Click the "output" button to see the full help page. Lists have lots of methods with weird-looking names like \_\_eq\_\_ and \_\_iadd\_\_. Don't worry too much about these for now. (You'll probably never call such methods directly. But they get called behind the scenes when we use syntax like indexing or comparison operators.) The most interesting methods are toward the bottom of the list (append, clear, copy, etc.).

## Tuples

Tuples are almost exactly the same as lists. They differ in just two ways.

**1:** The syntax for creating them uses (optional) parentheses rather than square brackets

In [34]:

t = (1, 2, 3)

In [35]:

t = 1, 2, 3 *# equivalent to above*

t

Out[35]:

(1, 2, 3)

**2:** They cannot be modified (they are immutable).

In [36]:

t[0] = 100

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

TypeError Traceback (most recent call last)

<ipython-input-36-e6cf7836e708> in <module>()

----> 1 t[0] = 100

TypeError: 'tuple' object does not support item assignment

Tuples are often used for functions that have multiple return values.

For example, the as\_integer\_ratio() method of float objects returns a numerator and a denominator in the form of a tuple:

In [37]:

x = 0.125

x.as\_integer\_ratio()

Out[37]:

(1, 8)

These multiple return values can be individually assigned as follows:

In [38]:

numerator, denominator = x.as\_integer\_ratio()

print(numerator / denominator)

0.125

Finally we have some insight into this classic Stupid Python Trick™ for swapping two variables!

In [39]:

a = 1

b = 0

a, b = b, a

print(a, b)

0 1

# **Your turn!**

Head over to [the Exercises notebook](https://www.kaggle.com/kernels/fork/961685) to get some hands-on practice working with lists.

# **DAY**

# **Loops**

Loops are a way to repeatedly execute some code. Here's an example:

In [1]:

planets = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']

for planet **in** planets:

print(planet, end=' ') *# print all on same line*

Mercury Venus Earth Mars Jupiter Saturn Uranus Neptune

The for loop specifies

* the variable name to use (in this case, planet)
* the set of values to loop over (in this case, planets)

You use the word "in" to link them together.

The object to the right of the "in" can be any object that supports iteration. Basically, if it can be thought of as a group of things, you can probably loop over it. In addition to lists, we can iterate over the elements of a tuple:

In [2]:

multiplicands = (2, 2, 2, 3, 3, 5)

product = 1

for mult **in** multiplicands:

product = product \* mult

product

Out[2]:

360

You can even loop through each character in a string:

In [3]:

s = 'steganograpHy is the practicE of conceaLing a file, message, image, or video within another fiLe, message, image, Or video.'

msg = ''

*# print all the uppercase letters in s, one at a time*

for char **in** s:

if char.isupper():

print(char, end='')

HELLO

### range()

range() is a function that returns a sequence of numbers. It turns out to be very useful for writing loops.

For example, if we want to repeat some action 5 times:

In [4]:

for i **in** range(5):

print("Doing important work. i =", i)

Doing important work. i = 0

Doing important work. i = 1

Doing important work. i = 2

Doing important work. i = 3

Doing important work. i = 4

## while loops

The other type of loop in Python is a while loop, which iterates until some condition is met:

In [5]:

i = 0

while i < 10:

print(i, end=' ')

i += 1

0 1 2 3 4 5 6 7 8 9

The argument of the while loop is evaluated as a boolean statement, and the loop is executed until the statement evaluates to False.

## List comprehensions

List comprehensions are one of Python's most beloved and unique features. The easiest way to understand them is probably to just look at a few examples:

In [6]:

squares = [n\*\*2 for n **in** range(10)]

squares

Out[6]:

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

Here's how we would do the same thing without a list comprehension:

In [7]:

squares = []

for n **in** range(10):

squares.append(n\*\*2)

squares

Out[7]:

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

We can also add an if condition:

In [8]:

short\_planets = [planet for planet **in** planets if len(planet) < 6]

short\_planets

Out[8]:

['Venus', 'Earth', 'Mars']

(If you're familiar with SQL, you might think of this as being like a "WHERE" clause)

Here's an example of filtering with an if condition and applying some transformation to the loop variable:

In [9]:

*# str.upper() returns an all-caps version of a string*

loud\_short\_planets = [planet.upper() + '!' for planet **in** planets if len(planet) < 6]

loud\_short\_planets

Out[9]:

['VENUS!', 'EARTH!', 'MARS!']

People usually write these on a single line, but you might find the structure clearer when it's split up over 3 lines:

In [10]:

[

planet.upper() + '!'

for planet **in** planets

if len(planet) < 6

]

Out[10]:

['VENUS!', 'EARTH!', 'MARS!']

(Continuing the SQL analogy, you could think of these three lines as SELECT, FROM, and WHERE)

The expression on the left doesn't technically have to involve the loop variable (though it'd be pretty unusual for it not to). What do you think the expression below will evaluate to? Press the 'output' button to check.

Hide

In [11]:

[32 for planet **in** planets]

Out[11]:

[32, 32, 32, 32, 32, 32, 32, 32]

List comprehensions combined with functions like min, max, and sum can lead to impressive one-line solutions for problems that would otherwise require several lines of code.

For example, compare the following two cells of code that do the same thing.

In [12]:

def count\_negatives(nums):

*"""Return the number of negative numbers in the given list.*

*>>> count\_negatives([5, -1, -2, 0, 3])*

*2*

*"""*

n\_negative = 0

for num **in** nums:

if num < 0:

n\_negative = n\_negative + 1

return n\_negative

Here's a solution using a list comprehension:

In [13]:

def count\_negatives(nums):

return len([num for num **in** nums if num < 0])

Much better, right?

Well if all we care about is minimizing the length of our code, this third solution is better still!

In [14]:

def count\_negatives(nums):

*# Reminder: in the "booleans and conditionals" exercises, we learned about a quirk of*

*# Python where it calculates something like True + True + False + True to be equal to 3.*

return sum([num < 0 for num **in** nums])

Which of these solutions is the "best" is entirely subjective. Solving a problem with less code is always nice, but it's worth keeping in mind the following lines from [The Zen of Python](https://en.wikipedia.org/wiki/Zen_of_Python):

Readability counts.  
Explicit is better than implicit.

So, use these tools to make compact readable programs. But when you have to choose, favor code that is easy for others to understand.

# **Your Turn**

Try the [hands-on exercise](https://www.kaggle.com/kernels/fork/961955) with loops and list comprehensions

# **DAY 6**

This lesson will be a double-shot of essential Python types: **strings** and **dictionaries**.

# **Strings**

One place where the Python language really shines is in the manipulation of strings. This section will cover some of Python's built-in string methods and formatting operations.

Such string manipulation patterns come up often in the context of data science work, and is one big perk of Python in this context.

## String syntax

You've already seen plenty of strings in examples during the previous lessons, but just to recap, strings in Python can be defined using either single or double quotations. They are functionally equivalent.

In [1]:

x = 'Pluto is a planet'

y = "Pluto is a planet"

x == y

Out[1]:

True

Double quotes are convenient if your string contains a single quote character (e.g. representing an apostrophe).

Similarly, it's easy to create a string that contains double-quotes if you wrap it in single quotes:

In [2]:

print("Pluto's a planet!")

print('My dog is named "Pluto"')

Pluto's a planet!

My dog is named "Pluto"

If we try to put a single quote character inside a single-quoted string, Python gets confused:

In [3]:

'Pluto's a planet!'

File "<ipython-input-3-a43631749f52>", line 1

'Pluto's a planet!'

^

SyntaxError: invalid syntax

We can fix this by "escaping" the single quote with a backslash.

In [4]:

'Pluto**\'**s a planet!'

Out[4]:

"Pluto's a planet!"

The table below summarizes some important uses of the backslash character.

| What you type... | What you get | example | print(example) |
| --- | --- | --- | --- |
| \' | ' | 'What\'s up?' | What's up? |
| \" | " | "That's \"cool\"" | That's "cool" |
| \\ | \ | "Look, a mountain: /\\" | Look, a mountain: /\ |
| \n |  | "1\n2 3" | 1 2 3 |

The last sequence, \n, represents the newline character. It causes Python to start a new line.

In [5]:

hello = "hello**\n**world"

print(hello)

hello

world

In addition, Python's triple quote syntax for strings lets us include newlines literally (i.e. by just hitting 'Enter' on our keyboard, rather than using the special '\n' sequence). We've already seen this in the docstrings we use to document our functions, but we can use them anywhere we want to define a string.

In [6]:

triplequoted\_hello = """hello

world"""

print(triplequoted\_hello)

triplequoted\_hello == hello

hello

world

Out[6]:

True

The print() function automatically adds a newline character unless we specify a value for the keyword argument end other than the default value of '\n':

In [7]:

print("hello")

print("world")

print("hello", end='')

print("pluto", end='')

hello

world

hellopluto

## Strings are sequences

Strings can be thought of as sequences of characters. Almost everything we've seen that we can do to a list, we can also do to a string.

In [8]:

*# Indexing*

planet = 'Pluto'

planet[0]

Out[8]:

'P'

In [9]:

*# Slicing*

planet[-3:]

Out[9]:

'uto'

In [10]:

*# How long is this string?*

len(planet)

Out[10]:

5

In [11]:

*# Yes, we can even loop over them*

[char+'! ' for char **in** planet]

Out[11]:

['P! ', 'l! ', 'u! ', 't! ', 'o! ']

But a major way in which they differ from lists is that they are immutable. We can't modify them.

In [12]:

planet[0] = 'B'

*# planet.append doesn't work either*

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

TypeError Traceback (most recent call last)

<ipython-input-12-6ca42463b9f9> in <module>()

----> 1 planet[0] = 'B'

**2** # planet.append doesn't work either

TypeError: 'str' object does not support item assignment

## String methods

Like list, the type str has lots of very useful methods. I'll show just a few examples here.

In [13]:

*# ALL CAPS*

claim = "Pluto is a planet!"

claim.upper()

Out[13]:

'PLUTO IS A PLANET!'

In [14]:

*# all lowercase*

claim.lower()

Out[14]:

'pluto is a planet!'

In [15]:

*# Searching for the first index of a substring*

claim.index('plan')

Out[15]:

11

In [16]:

claim.startswith(planet)

Out[16]:

True

In [17]:

claim.endswith('dwarf planet')

Out[17]:

False

### Going between strings and lists: .split() and .join()

str.split() turns a string into a list of smaller strings, breaking on whitespace by default. This is super useful for taking you from one big string to a list of words.

In [18]:

words = claim.split()

words

Out[18]:

['Pluto', 'is', 'a', 'planet!']

Occasionally you'll want to split on something other than whitespace:

In [19]:

datestr = '1956-01-31'

year, month, day = datestr.split('-')

str.join() takes us in the other direction, sewing a list of strings up into one long string, using the string it was called on as a separator.

In [20]:

'/'.join([month, day, year])

Out[20]:

'01/31/1956'

In [21]:

*# Yes, we can put unicode characters right in our string literals :)*

' 👏 '.join([word.upper() for word **in** words])

Out[21]:

'PLUTO 👏 IS 👏 A 👏 PLANET!'

### Building strings with .format()

Python lets us concatenate strings with the + operator.

In [22]:

planet + ', we miss you.'

Out[22]:

'Pluto, we miss you.'

If we want to throw in any non-string objects, we have to be careful to call str() on them first

In [23]:

position = 9

planet + ", you'll always be the " + position + "th planet to me."

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

TypeError Traceback (most recent call last)

<ipython-input-23-73295f9638cc> in <module>()

**1** position = 9

----> 2 planet + ", you'll always be the " + position + "th planet to me."

TypeError: must be str, not int

In [24]:

planet + ", you'll always be the " + str(position) + "th planet to me."

Out[24]:

"Pluto, you'll always be the 9th planet to me."

This is getting hard to read and annoying to type. str.format() to the rescue.

In [25]:

"**{}**, you'll always be the **{}**th planet to me.".format(planet, position)

Out[25]:

"Pluto, you'll always be the 9th planet to me."

So much cleaner! We call .format() on a "format string", where the Python values we want to insert are represented with {} placeholders.

Notice how we didn't even have to call str() to convert position from an int. format() takes care of that for us.

If that was all that format() did, it would still be incredibly useful. But as it turns out, it can do a lot more. Here's just a taste:

In [26]:

pluto\_mass = 1.303 \* 10\*\*22

earth\_mass = 5.9722 \* 10\*\*24

population = 52910390

*# 2 decimal points 3 decimal points, format as percent separate with commas*

"**{}** weighs about **{:.2}** kilograms (**{:.3%}** of Earth's mass). It is home to **{:,}** Plutonians.".format(

planet, pluto\_mass, pluto\_mass / earth\_mass, population,

)

Out[26]:

"Pluto weighs about 1.3e+22 kilograms (0.218% of Earth's mass). It is home to 52,910,390 Plutonians."

In [27]:

*# Referring to format() arguments by index, starting from 0*

s = """Pluto's a **{0}**.

No, it's a **{1}**.

**{0}**!

**{1}**!""".format('planet', 'dwarf planet')

print(s)

Pluto's a planet.

No, it's a dwarf planet.

planet!

dwarf planet!

You could probably write a short book just on str.format, so I'll stop here, and point you to [pyformat.info](https://pyformat.info/) and [the official docs](https://docs.python.org/3/library/string.html#formatstrings) for further reading.

# **Dictionaries**

Dictionaries are a built-in Python data structure for mapping keys to values.

In [28]:

numbers = {'one':1, 'two':2, 'three':3}

In this case 'one', 'two', and 'three' are the **keys**, and 1, 2 and 3 are their corresponding values.

Values are accessed via square bracket syntax similar to indexing into lists and strings.

In [29]:

numbers['one']

Out[29]:

1

We can use the same syntax to add another key, value pair

In [30]:

numbers['eleven'] = 11

numbers

Out[30]:

{'one': 1, 'two': 2, 'three': 3, 'eleven': 11}

Or to change the value associated with an existing key

In [31]:

numbers['one'] = 'Pluto'

numbers

Out[31]:

{'one': 'Pluto', 'two': 2, 'three': 3, 'eleven': 11}

Python has dictionary comprehensions with a syntax similar to the list comprehensions we saw in the previous tutorial.

In [32]:

planets = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']

planet\_to\_initial = {planet: planet[0] for planet **in** planets}

planet\_to\_initial

Out[32]:

{'Mercury': 'M',

'Venus': 'V',

'Earth': 'E',

'Mars': 'M',

'Jupiter': 'J',

'Saturn': 'S',

'Uranus': 'U',

'Neptune': 'N'}

The in operator tells us whether something is a key in the dictionary

In [33]:

'Saturn' **in** planet\_to\_initial

Out[33]:

True

In [34]:

'Betelgeuse' **in** planet\_to\_initial

Out[34]:

False

A for loop over a dictionary will loop over its keys

In [35]:

for k **in** numbers:

print("**{}** = **{}**".format(k, numbers[k]))

one = Pluto

two = 2

three = 3

eleven = 11

We can access a collection of all the keys or all the values with dict.keys() and dict.values(), respectively.

In [36]:

*# Get all the initials, sort them alphabetically, and put them in a space-separated string.*

' '.join(sorted(planet\_to\_initial.values()))

Out[36]:

'E J M M N S U V'

The very useful dict.items() method lets us iterate over the keys and values of a dictionary simultaneously. (In Python jargon, an **item** refers to a key, value pair)

In [37]:

for planet, initial **in** planet\_to\_initial.items():

print("**{}** begins with **\"{}\"**".format(planet.rjust(10), initial))

Mercury begins with "M"

Venus begins with "V"

Earth begins with "E"

Mars begins with "M"

Jupiter begins with "J"

Saturn begins with "S"

Uranus begins with "U"

Neptune begins with "N"

To read a full inventory of dictionaries' methods, click the "output" button below to read the full help page, or check out the [official online documentation](https://docs.python.org/3/library/stdtypes.html#dict).

Hide

In [38]:

help(dict)

Help on class dict in module builtins:

class dict(object)

| dict() -> new empty dictionary

| dict(mapping) -> new dictionary initialized from a mapping object's

| (key, value) pairs

| dict(iterable) -> new dictionary initialized as if via:

| d = {}

| for k, v in iterable:

| d[k] = v

| dict(\*\*kwargs) -> new dictionary initialized with the name=value pairs

| in the keyword argument list. For example: dict(one=1, two=2)

|

| Methods defined here:

|

| \_\_contains\_\_(self, key, /)

| True if D has a key k, else False.

|

| \_\_delitem\_\_(self, key, /)

| Delete self[key].

|

| \_\_eq\_\_(self, value, /)

| Return self==value.

|

| \_\_ge\_\_(self, value, /)

| Return self>=value.

|

| \_\_getattribute\_\_(self, name, /)

| Return getattr(self, name).

|

| \_\_getitem\_\_(...)

| x.\_\_getitem\_\_(y) <==> x[y]

|

| \_\_gt\_\_(self, value, /)

| Return self>value.

|

| \_\_init\_\_(self, /, \*args, \*\*kwargs)

| Initialize self. See help(type(self)) for accurate signature.

|

| \_\_iter\_\_(self, /)

| Implement iter(self).

|

| \_\_le\_\_(self, value, /)

| Return self<=value.

|

| \_\_len\_\_(self, /)

| Return len(self).

|

| \_\_lt\_\_(self, value, /)

| Return self<value.

|

| \_\_ne\_\_(self, value, /)

| Return self!=value.

|

| \_\_new\_\_(\*args, \*\*kwargs) from builtins.type

| Create and return a new object. See help(type) for accurate signature.

|

| \_\_repr\_\_(self, /)

| Return repr(self).

|

| \_\_setitem\_\_(self, key, value, /)

| Set self[key] to value.

|

| \_\_sizeof\_\_(...)

| D.\_\_sizeof\_\_() -> size of D in memory, in bytes

|

| clear(...)

| D.clear() -> None. Remove all items from D.

|

| copy(...)

| D.copy() -> a shallow copy of D

|

| fromkeys(iterable, value=None, /) from builtins.type

| Returns a new dict with keys from iterable and values equal to value.

|

| get(...)

| D.get(k[,d]) -> D[k] if k in D, else d. d defaults to None.

|

| items(...)

| D.items() -> a set-like object providing a view on D's items

|

| keys(...)

| D.keys() -> a set-like object providing a view on D's keys

|

| pop(...)

| D.pop(k[,d]) -> v, remove specified key and return the corresponding value.

| If key is not found, d is returned if given, otherwise KeyError is raised

|

| popitem(...)

| D.popitem() -> (k, v), remove and return some (key, value) pair as a

| 2-tuple; but raise KeyError if D is empty.

|

| setdefault(...)

| D.setdefault(k[,d]) -> D.get(k,d), also set D[k]=d if k not in D

|

| update(...)

| D.update([E, ]\*\*F) -> None. Update D from dict/iterable E and F.

| If E is present and has a .keys() method, then does: for k in E: D[k] = E[k]

| If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v

| In either case, this is followed by: for k in F: D[k] = F[k]

|

| values(...)

| D.values() -> an object providing a view on D's values

|

| ----------------------------------------------------------------------

| Data and other attributes defined here:

|

| \_\_hash\_\_ = None

# **Your Turn**

Try the [hands-on exercise](https://www.kaggle.com/kernels/fork/962743) with strings and dictionaries

# **DAY 7**

In this lesson, I'll be talking about **imports** in Python, giving some tips for working with unfamiliar libraries (and the objects they return), and digging into the guts of Python just a bit to talk about **operator overloading**.

## Imports

So far we've talked about types and functions which are built-in to the language.

But one of the best things about Python (especially if you're a data scientist) is the vast number of high-quality custom libraries that have been written for it.

Some of these libraries are in the "standard library", meaning you can find them anywhere you run Python. Others libraries can be easily added, even if they aren't always shipped with Python.

Either way, we'll access this code with **imports**.

We'll start our example by importing math from the standard library.

In [1]:

import math

print("It's math! It has type **{}**".format(type(math)))

It's math! It has type <class 'module'>

math is a module. A module is just a collection of variables (a namespace, if you like) defined by someone else. We can see all the names in math using the built-in function dir().

In [2]:

print(dir(math))

['\_\_doc\_\_', '\_\_file\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']

We can access these variables using dot syntax. Some of them refer to simple values, like math.pi:

In [3]:

print("pi to 4 significant digits = **{:.4}**".format(math.pi))

pi to 4 significant digits = 3.142

But most of what we'll find in the module are functions, like math.log:

In [4]:

math.log(32, 2)

Out[4]:

5.0

Of course, if we don't know what math.log does, we can call help() on it:

In [5]:

help(math.log)

Help on built-in function log in module math:

log(...)

log(x[, base])

Return the logarithm of x to the given base.

If the base not specified, returns the natural logarithm (base e) of x.

We can also call help() on the module itself. This will give us the combined documentation for all the functions and values in the module (as well as a high-level description of the module). Click the "output" button to see the whole math help page.

Output

In [6]:

help(math)

### Other import syntax

If we know we'll be using functions in math frequently we can import it under a shorter alias to save some typing (though in this case "math" is already pretty short).

In [7]:

import math as mt

mt.pi

Out[7]:

3.141592653589793

You may have seen code that does this with certain popular libraries like Pandas, Numpy, Tensorflow, or Matplotlib. For example, it's a common convention to import numpy as np and import pandas as pd.

The as simply renames the imported module. It's equivalent to doing something like:

In [8]:

import math

mt = math

Wouldn't it be great if we could refer to all the variables in the math module by themselves? i.e. if we could just refer to pi instead of math.pi or mt.pi? Good news: we can do that.

In [9]:

from math import \*

print(pi, log(32, 2))

3.141592653589793 5.0

import \* makes all the module's variables directly accessible to you (without any dotted prefix).

Bad news: some purists might grumble at you for doing this.

Worse: they kind of have a point.

In [10]:

from math import \*

from numpy import \*

print(pi, log(32, 2))

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

TypeError Traceback (most recent call last)

<ipython-input-10-5045b296ad83> in <module>()

**1** from math import \*

**2** from numpy import \*

----> 3 print(pi, log(32, 2))

TypeError: return arrays must be of ArrayType

What the what? But it worked before!

These kinds of "star imports" can occasionally lead to weird, difficult-to-debug situations.

The problem in this case is that the math and numpy modules both have functions called log, but they have different semantics. Because we import from numpy second, its log overwrites (or "shadows") the log variable we imported from math.

A good compromise is to import only the specific things we'll need from each module:

In [11]:

from math import log, pi

from numpy import asarray

### Submodules

We've seen that modules contain variables which can refer to functions or values. Something to be aware of is that they can also have variables referring to other modules.

In [12]:

import numpy

print("numpy.random is a", type(numpy.random))

print("it contains names such as...",

dir(numpy.random)[-15:]

)

numpy.random is a <class 'module'>

it contains names such as... ['set\_state', 'shuffle', 'standard\_cauchy', 'standard\_exponential', 'standard\_gamma', 'standard\_normal', 'standard\_t', 'test', 'triangular', 'uniform', 'vonmises', 'wald', 'warnings', 'weibull', 'zipf']

So if we import numpy as above, then calling a function in the random"submodule" will require two dots.

In [13]:

*# Roll 10 dice*

rolls = numpy.random.randint(low=1, high=6, size=10)

rolls

Out[13]:

array([3, 4, 4, 1, 4, 1, 3, 1, 5, 1])

# **Oh the places you'll go, oh the objects you'll see**

So after 6 lessons, you're a pro with ints, floats, bools, lists, strings, and dicts (right?).

Even if that were true, it doesn't end there. As you work with various libraries for specialized tasks, you'll find that they define their own types which you'll have to learn to work with. For example, if you work with the graphing library matplotlib, you'll be coming into contact with objects it defines which represent Subplots, Figures, TickMarks, and Annotations. pandas functions will give you DataFrames and Series.

In this section, I want to share with you a quick survival guide for working with strange types.

## Three tools for understanding strange objects

In the cell above, we saw that calling a numpy function gave us an "array". We've never seen anything like this before (not in this course anyways). But don't panic: we have three familiar builtin functions to help us here.

**1: type()** (what is this thing?)

In [14]:

type(rolls)

Out[14]:

numpy.ndarray

**2: dir()** (what can I do with it?)

In [15]:

print(dir(rolls))

['T', '\_\_abs\_\_', '\_\_add\_\_', '\_\_and\_\_', '\_\_array\_\_', '\_\_array\_finalize\_\_', '\_\_array\_function\_\_', '\_\_array\_interface\_\_', '\_\_array\_prepare\_\_', '\_\_array\_priority\_\_', '\_\_array\_struct\_\_', '\_\_array\_ufunc\_\_', '\_\_array\_wrap\_\_', '\_\_bool\_\_', '\_\_class\_\_', '\_\_complex\_\_', '\_\_contains\_\_', '\_\_copy\_\_', '\_\_deepcopy\_\_', '\_\_delattr\_\_', '\_\_delitem\_\_', '\_\_dir\_\_', '\_\_divmod\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_float\_\_', '\_\_floordiv\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getitem\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_iadd\_\_', '\_\_iand\_\_', '\_\_ifloordiv\_\_', '\_\_ilshift\_\_', '\_\_imatmul\_\_', '\_\_imod\_\_', '\_\_imul\_\_', '\_\_index\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_int\_\_', '\_\_invert\_\_', '\_\_ior\_\_', '\_\_ipow\_\_', '\_\_irshift\_\_', '\_\_isub\_\_', '\_\_iter\_\_', '\_\_itruediv\_\_', '\_\_ixor\_\_', '\_\_le\_\_', '\_\_len\_\_', '\_\_lshift\_\_', '\_\_lt\_\_', '\_\_matmul\_\_', '\_\_mod\_\_', '\_\_mul\_\_', '\_\_ne\_\_', '\_\_neg\_\_', '\_\_new\_\_', '\_\_or\_\_', '\_\_pos\_\_', '\_\_pow\_\_', '\_\_radd\_\_', '\_\_rand\_\_', '\_\_rdivmod\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_rfloordiv\_\_', '\_\_rlshift\_\_', '\_\_rmatmul\_\_', '\_\_rmod\_\_', '\_\_rmul\_\_', '\_\_ror\_\_', '\_\_rpow\_\_', '\_\_rrshift\_\_', '\_\_rshift\_\_', '\_\_rsub\_\_', '\_\_rtruediv\_\_', '\_\_rxor\_\_', '\_\_setattr\_\_', '\_\_setitem\_\_', '\_\_setstate\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_sub\_\_', '\_\_subclasshook\_\_', '\_\_truediv\_\_', '\_\_xor\_\_', 'all', 'any', 'argmax', 'argmin', 'argpartition', 'argsort', 'astype', 'base', 'byteswap', 'choose', 'clip', 'compress', 'conj', 'conjugate', 'copy', 'ctypes', 'cumprod', 'cumsum', 'data', 'diagonal', 'dot', 'dtype', 'dump', 'dumps', 'fill', 'flags', 'flat', 'flatten', 'getfield', 'imag', 'item', 'itemset', 'itemsize', 'max', 'mean', 'min', 'nbytes', 'ndim', 'newbyteorder', 'nonzero', 'partition', 'prod', 'ptp', 'put', 'ravel', 'real', 'repeat', 'reshape', 'resize', 'round', 'searchsorted', 'setfield', 'setflags', 'shape', 'size', 'sort', 'squeeze', 'std', 'strides', 'sum', 'swapaxes', 'take', 'tobytes', 'tofile', 'tolist', 'tostring', 'trace', 'transpose', 'var', 'view']

In [16]:

*# What am I trying to do with this dice roll data? Maybe I want the average roll, in which case the "mean"*

*# method looks promising...*

rolls.mean()

Out[16]:

2.7

In [17]:

*# Or maybe I just want to get back on familiar ground, in which case I might want to check out "tolist"*

rolls.tolist()

Out[17]:

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

**3: help()** (tell me more)

In [18]:

*# That "ravel" attribute sounds interesting. I'm a big classical music fan.*

help(rolls.ravel)

Help on built-in function ravel:

ravel(...) method of numpy.ndarray instance

a.ravel([order])

Return a flattened array.

Refer to `numpy.ravel` for full documentation.

See Also

--------

numpy.ravel : equivalent function

ndarray.flat : a flat iterator on the array.

Output

In [19]:

*# Okay, just tell me everything there is to know about numpy.ndarray*

*# (Click the "output" button to see the novel-length output)*

help(rolls)

(Of course, you might also prefer to check out [the online docs](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.ndarray.html))

### Operator overloading

What's the value of the below expression?

In [20]:

[3, 4, 1, 2, 2, 1] + 10

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

TypeError Traceback (most recent call last)

<ipython-input-20-a2508fc27c2b> in <module>()

----> 1 [3, 4, 1, 2, 2, 1] + 10

TypeError: can only concatenate list (not "int") to list

What a silly question. Of course it's an error.

But what about...

In [21]:

rolls + 10

Out[21]:

array([13, 14, 14, 11, 14, 11, 13, 11, 15, 11])

We might think that Python strictly polices how pieces of its core syntax behave such as +, <, in, ==, or square brackets for indexing and slicing. But in fact, it takes a very hands-off approach. When you define a new type, you can choose how addition works for it, or what it means for an object of that type to be equal to something else.

The designers of lists decided that adding them to numbers wasn't allowed. The designers of numpy arrays went a different way (adding the number to each element of the array).

Here are a few more examples of how numpy arrays interact unexpectedly with Python operators (or at least differently from lists).

In [22]:

*# At which indices are the dice less than or equal to 3?*

rolls <= 3

Out[22]:

array([ True, False, False, True, False, True, True, True, False,

True])

In [23]:

xlist = [[1,2,3],[2,4,6],]

*# Create a 2-dimensional array*

x = numpy.asarray(xlist)

print("xlist = **{}\n**x =**\n{}**".format(xlist, x))

xlist = [[1, 2, 3], [2, 4, 6]]

x =

[[1 2 3]

[2 4 6]]

In [24]:

*# Get the last element of the second row of our numpy array*

x[1,-1]

Out[24]:

6

In [25]:

*# Get the last element of the second sublist of our nested list?*

xlist[1,-1]

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

TypeError Traceback (most recent call last)

<ipython-input-25-e2f4c7f35788> in <module>()

**1** # Get the last element of the second sublist of our nested list?

----> 2 xlist[1,-1]

TypeError: list indices must be integers or slices, not tuple

numpy's ndarray type is specialized for working with multi-dimensional data, so it defines its own logic for indexing, allowing us to index by a tuple to specify the index at each dimension.

### When does 1 + 1 not equal 2?

Things can get weirder than this. You may have heard of (or even used) tensorflow, a Python library popularly used for deep learning. It makes extensive use of operator overloading.

In [26]:

import tensorflow as tf

*# Create two constants, each with value 1*

a = tf.constant(1)

b = tf.constant(1)

*# Add them together to get...*

a + b

Out[26]:

<tf.Tensor 'add:0' shape=() dtype=int32>

a + b isn't 2, it is (to quote tensorflow's documentation)...

a symbolic handle to one of the outputs of an Operation. It does not hold the values of that operation's output, but instead provides a means of computing those values in a TensorFlow tf.Session.

It's important just to be aware of the fact that this sort of thing is possible and that libraries will often use operator overloading in non-obvious or magical-seeming ways.

Understanding how Python's operators work when applied to ints, strings, and lists is no guarantee that you'll be able to immediately understand what they do when applied to a tensorflow Tensor, or a numpy ndarray, or a pandas DataFrame.

Once you've had a little taste of DataFrames, for example, an expression like the one below starts to look appealingly intuitive:

*# Get the rows with population over 1m in South America*

df[(df['population'] > 10\*\*6) & (df['continent'] == 'South America')]

But why does it work? The example above features something like **5** different overloaded operators. What's each of those operations doing? It can help to know the answer when things start going wrong.

#### **Curious how it all works?**

Have you ever called help() or dir() on an object and wondered what the heck all those names with the double-underscores were?

In [27]:

print(dir(list))

['\_\_add\_\_', '\_\_class\_\_', '\_\_contains\_\_', '\_\_delattr\_\_', '\_\_delitem\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getitem\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_iadd\_\_', '\_\_imul\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_iter\_\_', '\_\_le\_\_', '\_\_len\_\_', '\_\_lt\_\_', '\_\_mul\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_reversed\_\_', '\_\_rmul\_\_', '\_\_setattr\_\_', '\_\_setitem\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']

This turns out to be directly related to operator overloading.

When Python programmers want to define how operators behave on their types, they do so by implementing methods with special names beginning and ending with 2 underscores such as \_\_lt\_\_, \_\_setattr\_\_, or \_\_contains\_\_. Generally, names that follow this double-underscore format have a special meaning to Python.

So, for example, the expression x in [1, 2, 3] is actually calling the list method \_\_contains\_\_ behind-the-scenes. It's equivalent to (the much uglier) [1, 2, 3].\_\_contains\_\_(x).

If you're curious to learn more, you can check out [Python's official documentation](https://docs.python.org/3.4/reference/datamodel.html#special-method-names), which describes many, many more of these special "underscores" methods.

We won't be defining our own types in these lessons (if only there was time!), but I hope you'll get to experience the joys of defining your own wonderful, weird types later down the road.

# **Your turn!**

Head over to [the very last Exercises notebook](https://www.kaggle.com/kernels/fork/1164833) for one more round of coding questions involving imports, working with unfamiliar objects, and, of course, more gambling.