# Introduction to Python #3 Functions and Packages Familiar functions

Out of the box, Python offers a bunch of built-in functions to make your life as a data scientist easier. You already know two such functions: [**print()**](https://docs.python.org/3/library/functions.html#print) and [**type()**](https://docs.python.org/3/library/functions.html#type). You've also used the functions [**str()**](https://docs.python.org/3/library/functions.html#func-str), [**int()**](https://docs.python.org/3/library/functions.html#int), [**bool()**](https://docs.python.org/3/library/functions.html#bool) and [**float()**](https://docs.python.org/3/library/functions.html#float) to switch between data types. These are built-in functions as well.

Calling a function is easy. To get the type of 3.0 and store the output as a new variable, result, you can use the following:

result = type(3.0)

The general recipe for calling functions and saving the result to a variable is thus:

output = function\_name(input)

**Instructions**

**100 XP**

* Use [**print()**](https://docs.python.org/3/library/functions.html#print) in combination with [**type()**](https://docs.python.org/3/library/functions.html#type) to print out the type of var1.
* Use **[len()](https://docs.python.org/3/library/functions.html" \l "len" \t "_blank)** to get the length of the list var1. Wrap it in a [**print()**](https://docs.python.org/3/library/functions.html#print) call to directly print it out.
* Use [**int()**](https://docs.python.org/3/library/functions.html#int) to convert var2 to an integer. Store the output as out2.

Script.py  
01 # Create variables var1 and var2

02 var1 = [1, 2, 3, 4]

03 var2 = True

04

05 # Print out type of var1

06 print(type(var1))

07

08 # Print out length of var1

09 print(len(var1))

10

11 # Convert var2 to an integer: out2

12 out2 = int(var2)

IPythrn Shell  
In [1]: # Create variables var1 and var2

var1 = [1, 2, 3, 4]

var2 = True

# Print out type of var1

print(type(var1))

# Print out length of var1

print(len(var1))

# Convert var2 to an integer: out2

out2 = int(var2)

<class 'list'>

4

In [2]:

**Help!**

Maybe you already know the name of a Python function, but you still have to figure out how to use it. Ironically, you have to ask for information about a function with another function: [**help()**](https://docs.python.org/3/library/functions.html#help). In IPython specifically, you can also use ? before the function name.

To get help on the [**max()**](https://docs.python.org/3/library/functions.html#max) function, for example, you can use one of these calls:

help(max)

?max

Use the Shell on the right to open up the documentation on [**complex()**](https://docs.python.org/3/library/functions.html#complex). Which of the following statements is true?

**Instructions**

**50 XP**

**Instructions**

**50 XP**

**Possible Answers**

* 

[**complex()**](https://docs.python.org/3/library/functions.html#complex) takes exactly two arguments: real and [, imag].

* 

[**complex()**](https://docs.python.org/3/library/functions.html#complex) takes two arguments: real and imag. Both these arguments are required.

* 

[**complex()**](https://docs.python.org/3/library/functions.html#complex) takes two arguments: real and imag. real is a required argument, imag is an optional argument.

* 

[**complex()**](https://docs.python.org/3/library/functions.html#complex) takes two arguments: real and imag. If you don't specify imag, it is set to 1 by Python.

IPython Shell  
In [1]: help(complex)

Help on class complex in module builtins:

class complex(object)

| complex(real[, imag]) -> complex number

|

| Create a complex number from a real part and an optional imaginary part.

| This is equivalent to (real + imag\*1j) where imag defaults to 0.

|

| Methods defined here:

|

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

| abs(self)

|

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

| Return self+value.

|

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

| self != 0

|

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

| Return divmod(self, value).

|

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

| Return self==value.

|

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

| float(self)

|

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

| Return self//value.

|

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

| complex.\_\_format\_\_() -> str

|

| Convert to a string according to format\_spec.

|

| \_\_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).

|

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

| int(self)

|

| \_\_le\_\_(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.

|

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

| +self

|

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

| Return pow(self, value, mod).

|

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

| Return value+self.

|

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

| Return divmod(value, self).

|

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

| Return repr(self).

|

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

| Return value//self.

|

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

| Return value%self.

|

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

| Return value\*self.

|

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

| Return pow(value, self, mod).

|

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

| Return value-self.

|

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

| Return value/self.

|

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

| Return str(self).

|

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

| Return self-value.

|

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

| Return self/value.

|

| conjugate(...)

| complex.conjugate() -> complex

|

| Return the complex conjugate of its argument. (3-4j).conjugate() == 3+4j.

|

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

| Data descriptors defined here:

|

| imag

| the imaginary part of a complex number

|

| real

| the real part of a complex number

In [2]:

**Multiple arguments**

In the previous exercise, the square brackets around imag in the documentation showed us that the imagargument is optional. But Python also uses a different way to tell users about arguments being optional.

Have a look at the documentation of [**sorted()**](https://docs.python.org/3/library/functions.html#sorted) by typing help(sorted) in the IPython Shell.

You'll see that [**sorted()**](https://docs.python.org/3/library/functions.html#sorted) takes three arguments: iterable, key and reverse.

key=None means that if you don't specify the keyargument, it will be None. reverse=False means that if you don't specify the reverse argument, it will be False.

In this exercise, you'll only have to specify iterable and reverse, not key. The first input you pass to [**sorted()**](https://docs.python.org/3/library/functions.html#sorted) will be matched to the iterable argument, but what about the second input? To tell Python you want to specify reverse without changing anything about key, you can use =:

sorted(\_\_\_, reverse = \_\_\_)

Two lists have been created for you on the right. Can you paste them together and sort them in descending order?

Note: For now, we can understand an ***[iterable](https://docs.python.org/2/glossary.html" \l "term-iterable" \t "_blank)*** as being any collection of objects, e.g. a List.

**Instructions**

**100 XP**

* Use + to merge the contents of first and secondinto a new list: full.
* Call [**sorted()**](https://docs.python.org/3/library/functions.html#sorted) on full and specify the reverseargument to be True. Save the sorted list as full\_sorted.
* Finish off by printing out full\_sorted.

Script.py  
# Create lists first and second

first = [11.25, 18.0, 20.0]

second = [10.75, 9.50]

# Paste together first and second: full

full = first + second

# Sort full in descending order: full\_sorted

full\_sorted = sorted(full, reverse = True)

# Print out full\_sorted

print(full\_sorted)

IPython Shell  
In [1]: # Create lists first and second

first = [11.25, 18.0, 20.0]

second = [10.75, 9.50]

# Paste together first and second: full

full = first + second

In [2]:

**String Methods**

Strings come with a bunch of methods. Follow the instructions closely to discover some of them. If you want to discover them in more detail, you can always type help(str) in the IPython Shell.

A string place has already been created for you to experiment with.

**Instructions**

**100 XP**

* Use the [**upper()**](https://docs.python.org/3/library/stdtypes.html#str.upper) method on place and store the result in place\_up. Use the syntax for calling methods that you learned in the previous video.
* Print out place and place\_up. Did both change?
* Print out the number of o's on the variable place by calling [**count()**](https://docs.python.org/3/library/stdtypes.html#str.count) on place and passing the letter 'o' as an input to the method. We're talking about the variable place, not the word "place"!

Script.py  
01 # string to experiment with: place

02 place = "poolhouse"

03

04 # Use upper() on place: place\_up

05 place\_up = place.upper()

06

07 # Print out place and place\_up

08 print(place)

09 print(place\_up)

10

11 # Print out the number of o's in place

12 print(place.count("o"))

IPython Shell  
In [1]: # string to experiment with: place

place = "poolhouse"

# Use upper() on place: place\_up

place\_up = place.upper()

# Print out place and place\_up

print(place)

print(place\_up)

# Print out the number of o's in place

print(place.count("o"))

poolhouse

POOLHOUSE

3

In [2]:

**List Methods**

Strings are not the only Python types that have methods associated with them. Lists, floats, integers and booleans are also types that come packaged with a bunch of useful methods. In this exercise, you'll be experimenting with:

* [**index()**](https://docs.python.org/3/library/stdtypes.html#str.index), to get the index of the first element of a list that matches its input and
* [**count()**](https://docs.python.org/3/library/stdtypes.html#str.count), to get the number of times an element appears in a list.

You'll be working on the list with the area of different parts of a house: areas.

**Instructions**

**100 XP**

* Use the [**index()**](https://docs.python.org/3/library/stdtypes.html#str.index) method to get the index of the element in areas that is equal to 20.0. Print out this index.
* Call [**count()**](https://docs.python.org/3/library/stdtypes.html#str.count) on areas to find out how many times 9.50 appears in the list. Again, simply print out this number.

Script.py  
1 # Create list areas

2 areas = [11.25, 18.0, 20.0, 10.75, 9.50]

3

4 # Print out the index of the element 20.0

5 print(areas.index(20))

6

7 # Print out how often 9.50 appears in areas

8 print(areas.count(9.50))

IPython Shell  
In [1]: # Create list areas

areas = [11.25, 18.0, 20.0, 10.75, 9.50]

# Print out the index of the element 20.0

print(areas.index(20))

# Print out how often 9.50 appears in areas

print(areas.count(9.50))

2

1

In [2]:

**List Methods (2)**

Most list methods will change the list they're called on. Examples are:

* [**append()**](https://docs.python.org/3/library/stdtypes.html#typesseq-mutable), that adds an element to the list it is called on,
* [**remove()**](https://docs.python.org/3/library/stdtypes.html#typesseq-mutable), that removes the first element of a list that matches the input, and
* [**reverse()**](https://docs.python.org/3/library/stdtypes.html#typesseq-mutable), that reverses the order of the elements in the list it is called on.

You'll be working on the list with the area of different parts of the house: areas.

**Instructions**

**100 XP**

* Use [**append()**](https://docs.python.org/3/library/stdtypes.html#typesseq-mutable) twice to add the size of the poolhouse and the garage again: 24.5 and 15.45, respectively. Make sure to add them in this order.
* Print out areas
* Use the [**reverse()**](https://docs.python.org/3/library/stdtypes.html#typesseq-mutable) method to reverse the order of the elements in areas.
* Print out areas once more.

Script.py  
01 # Create list areas

02 areas = [11.25, 18.0, 20.0, 10.75, 9.50]

03

04 # Use append twice to add poolhouse and garage size

05 areas.append(24.5)

06 areas.append(15.45)

07

08 # Print out areas

09 print(areas)

10

11 # Reverse the orders of the elements in areas

12 areas.reverse()

13

14 # Print out areas

15 print(areas)

IPython Shell  
In [1]: # Create list areas

areas = [11.25, 18.0, 20.0, 10.75, 9.50]

# Use append twice to add poolhouse and garage size

areas.append(24.5)

areas.append(15.45)

# Print out areas

print(areas)

# Reverse the orders of the elements in areas

areas.reverse()

# Print out areas

print(areas)

[11.25, 18.0, 20.0, 10.75, 9.5, 24.5, 15.45]

[15.45, 24.5, 9.5, 10.75, 20.0, 18.0, 11.25]

**Import package**

As a data scientist, some notions of geometry never hurt. Let's refresh some of the basics.

For a fancy clustering algorithm, you want to find the circumference, CC, and area, AA, of a circle. When the radius of the circle is r, you can calculate CC and AA as:

C=2πrC=2πr

A=πr2A=πr2

To use the constant pi, you'll need the math package. A variable r is already coded in the script. Fill in the code to calculate C and A and see how the [**print()**](https://docs.python.org/3/library/functions.html#print)functions create some nice printouts.

**Instructions**

**100 XP**

* Import the math package. Now you can access the constant pi with math.pi.
* Calculate the circumference of the circle and store it in C.
* Calculate the area of the circle and store it in A.

Script.py  
01 # Definition of radius

02 r = 0.43

03

04 # Import the math package

05 import math

06

07 # Calculate C

08 C = math.pi \* 2 \* r

09

10 # Calculate A

11 A = math.pi \* r \* r

12

13 # Build printout

14 print("Circumference: " + str(C))

15 print("Area: " + str(A))

IPython Shell  
In [1]: # Definition of radius

r = 0.43

# Import the math package

import math

# Calculate C

C = math.pi \* 2 \* r

# Calculate A

A = math.pi \* r \* r

# Build printout

print("Circumference: " + str(C))

print("Area: " + str(A))

Circumference: 2.701769682087222

Area: 0.5808804816487527

**Selective import**

General imports, like import math, make **all**functionality from the math package available to you. However, if you decide to only use a specific part of a package, you can always make your import more selective:

from math import pi

Let's say the Moon's orbit around planet Earth is a perfect circle, with a radius r (in km) that is defined in the script.

**Instructions**

**100 XP**

* Perform a selective import from the math package where you only import the radians function.
* Calculate the distance travelled by the Moon over 12 degrees of its orbit. Assign the result to dist. You can calculate this as r \* phi, where r is the radius and phi is the angle in radians. To convert an angle in degrees to an angle in radians, use the [**radians()**](https://docs.python.org/3/library/math.html#math.radians)function, which you just imported.
* Print out dist.

Script.py  
01 # Definition of radius

02 r = 192500

03

04 # Import radians function of math package

05 from math import radians

06

07 # Travel distance of Moon over 12 degrees. Store in dist.

08 dist = r \* radians(12)

09

10 # Print out dist

11 print(dist)

IPython Shell  
In [1]: # Definition of radius

r = 192500

# Import radians function of math package

from math import radians

# Travel distance of Moon over 12 degrees. Store in dist.

dist = r \* radians(12)

# Print out dist

print(dist)

40317.10572106901

In [2]: dist = r \* radians(12)

# In [3]: Different ways of importing

There are several ways to import packages and modules into Python. Depending on the import call, you'll have to use different Python code.

Suppose you want to use the function [**inv()**](http://docs.scipy.org/doc/numpy-1.10.0/reference/generated/numpy.linalg.inv.html), which is in the linalgsubpackage of the scipy package. You want to be able to use this function as follows:

my\_inv([[1,2], [3,4]])

Which import statement will you need in order to run the above code without an error?

**Instructions**

**50 XP**

**Possible Answers**

* 

import scipy

* 

import scipy.linalg

* 

from scipy.linalg import my\_inv

* 

from scipy.linalg import inv as my\_inv