

Python for Labs/Lectures

Lecture 0: Introduction to Python and Jupyter Notebooks

Nicholas Lee-Hone

July 2019

```
In [1]: %%HTML
<style>
.slides td, .slides th {
    font-size: 40px
}

.slides img{
    margin: 0 auto !important;
    display: block;
}
</style>
```

This set of lectures is designed to get you up to speed on how to use Python in a teaching environment.

- Lecture 0: Introduction to Python and Jupyter (with Syzygy)
- Lecture 1: Numeric computation (using Numpy)
- Lecture 2: Scientific computation (using Scipy)
- Lecture 3: Plotting (using Matplotlib)
- Lecture 4: Fitting to data (using Scipy)
- Lecture 5: Other topics (Monte Carlo, using notebooks in lectures, custom topics?)

- Why Python?
- Notebooks
- Using Python as a calculator
- Variables
 - Numbers
 - Strings
 - Collections (list, tuple, dictionary, set)
 - Ranges
- Conditionals (if/elif/else)
- Looping (for/while loops)
- Functions

There are six modules that will take you from complete beginner to knowing enough to use Python in a course.

Why Python?

Pros	Cons
Simple to learn	Verbose
Interpreted (run step by step)	Slow (some workarounds exist)
Large package repository	Package quality and documentation varies significantly
Jupyter Notebooks	...

Python is simple to learn, but slightly more verbose than some other programming languages. However, for the purpose of teaching this is not a bad thing in general.

It is an interpreted language, which means that you can run small blocks of code, see the results, and then re-run the code with slight modifications, without re-compiling the whole program or running it from the beginning.

Python's open source package repository contains thousands of packages that range from numerical computation, machine learning, and lab instrument automation, to running web services and websites. Since the packages are maintained by the open source community (with no user fees for the most part), the quality of the packages and their documentation varies significantly. However the packages we will be using are mature and decently documented for the most part.

Finally one of the main selling points of Python is the Jupyter Notebook environment, which allows users to write code alongside documentation, images, and even LaTeX.

Go to <https://sfu.syzygy.ca/> (<https://sfu.syzygy.ca/>) and enter your credentials after clicking on the red house icon.



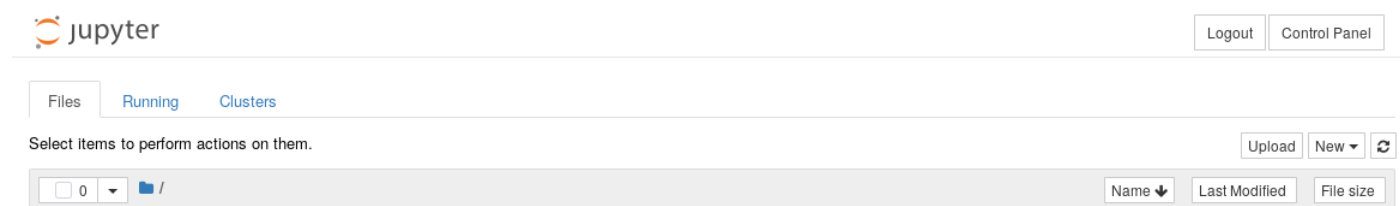
Sign in with your Simon Fraser University account



[Terms and conditions](#) / [Privacy Policy](#)

To start things off we need to log into Syzygy, which provides an already-installed Python and notebook environment. This service is available to all students at SFU and will probably be the easiest way to have students work with Python. The only requirement to access Python through syzygy is a working internet connection.

After entering your credentials you should see the Jupyter Notebook home screen:



Now we can create a folder in which to do some work. We'll start by doing this, and then move on to downloading some already-prepared notes. The already-prepared notes will be the main way that you can distribute Python related course material to the students.

1. Click the 'New' button on the right hand side of the screen and then 'Folder'.
2. Rename the folder by clicking on the small square to the left of the folder and then selecting 'Rename' in the toolbar above. This is a bit clunky but is unfortunately what we have to deal with for now.
3. Now click the folder to go into it. To go back, click the blue folder beside the folder name.
4. In the newly created folder create a new Python notebook by clicking the 'New' button again and selecting 'Python 3'.
5. Click the newly created notebook to open it.

Running notebook cells

In the first cell, enter a simple equation like the one shown below and press 'Shift+Enter' to run the code cell

```
In [2]: # Comments start with the pound symbol  
# The output shows up in the 'Out' part of the cell  
2 + 3
```

Out[2]: 5

```
In [3]: # Run two calculations in the same cell.  
# Note that only the output of the last line is displayed.  
2*3  
2 + 3
```

Out[3]: 5

```
In [4]: # Use the print() function to display values.  
# The print() output is not in the 'Out' part of the cell.  
print(2*3)  
2 + 3
```

6

Out[4]: 5

```
In [5]: # Show the results of the two calculations.  
# Note that there is no 'Out' cell  
print(2*3)  
print(2 + 3)
```

6

5

Download the lecture notebook

Enter the following URL into your navigation bar to download the first lecture content into your syzygy environment:

<https://sfu.syzygy.ca/jupyter/hub/user-redirect/git-pull?>

[repo=https://github.com/nleehone/PythonLectures.git&branch=master](https://github.com/nleehone/PythonLectures.git&branch=master) (<https://sfu.syzygy.ca/jupyter/hub/user-redirect/git-pull?repo=https://github.com/nleehone/PythonLectures.git&branch=master>)

This is one of the ways you can distribute course materials to students. Have the material hosted on a SFU git server and then just give them the link.

Simple math

<https://docs.python.org/3.6/tutorial/introduction.html> (<https://docs.python.org/3.6/tutorial/introduction.html>)

```
In [6]: # Division  
print(26/7)  
print(-26/7)  
  
# Integer division (floor)  
print(26//7)  
print(-26//7)
```

3.7142857142857144

-3.7142857142857144

3

-4

```
In [7]: # Power
print(2**3)
print(3.14**3)
```

```
8
30.959144000000002
```

```
In [8]: # Modulo (remainder)
print(22.3 % 9)
print(18 % 5)
```

```
4.300000000000001
3
```

Boolean logic

```
In [9]: # Logic operators
print(3 < 3)
print(3 <= 3)
print(3 > 3)
print(3 >= 3)
```

```
False
True
False
True
```

```
In [10]: # Equality?
print(3 != 4)
print(3 == 4)
print(not 3 == 4) # Not is the negation operator
```

```
True
False
True
```

```
In [11]: # Boolean logic
print(True and False)
print(True or False)
print(True and True)
```

```
False
True
True
```

```
In [12]: # Be EXTREMELY careful when checking floating point equality  
# It should almost NEVER be done  
print("0.1 + 0.2 = {}".format(0.1 + 0.2))  
print("{}".format(0.3))  
0.1 + 0.2 == 0.3
```

```
0.1 + 0.2 = 0.30000000000000004  
0.3
```

Out[12]: False

```
In [13]: # The best you can do is check that they are close within some thresh  
old  
print(abs((0.1 + 0.2) - 0.3))  
print(abs((0.1 + 0.2) - 0.3) < 1e-8) # Close to within some specified  
tolerance
```

```
5.551115123125783e-17  
True
```







Operator precedence

Level	Operator	Description
18	()	Grouping
17	f()	Function call
16	[index:index]	Slicing
15	[]	Array Subscription
14	**	Exponential
13	~	Bitwise NOT
12	+ -	Unary plus / minus
11	* / %	Multiplication Division Modulo
10	+ -	Addition / Subtraction
9	<< >>	Bitwise Left Shift Bitwise Right Shift
8	&	Bitwise AND
7	^	Bitwise XOR
6		Bitwise OR
5	in , not in, is , is not <, <=, >, >= == !=	Membership Relational Equality Inequality
4	not	Boolean NOT
3	and	Boolean AND
2	or	Boolean OR
1	lambda	Lambda Expression

Operators are evaluated from the highest precedence (level) to lowest. This ensures compatibility with the usual order we expect mathematical equations to be evaluated in.

Variables

- <https://docs.python.org/3.6/tutorial/introduction.html#using-python-as-a-calculator>
(<https://docs.python.org/3.6/tutorial/introduction.html#using-python-as-a-calculator>)
- <http://phy224.ca/02-variables/index.html> (<http://phy224.ca/02-variables/index.html>)

C/C++, Java	Python
<pre>int a = 1;</pre> 	<pre>a = 1</pre> 
<pre>a = 2;</pre> 	<pre>a = 2</pre> 
<pre>int b = a;</pre> 	<pre>b = a</pre> 

Variable assignment in Python is slightly different from what you might be used to if you come from C or Java. Variables do not represent blocks of memory where information can be stored, but are instead references to a memory location. We can move that reference around between memory locations.

This also means that variable type can and often does change during program execution. A variable name can be re-used many times for different types of data.

Use single equals sign (=) to store a value in a variable

```
In [14]: # Store values in variables using the equals sign  
a = 2  
b = 3  
a + b
```

Out[14]: 5

```
In [15]: # Store the result of a calculation in a cell using the equals sign  
c = a + b  
c
```

Out[15]: 5

```
In [16]: # Now go and change the value of a  
a = 10  
# Note that if we show the value of c again, it has not changed  
print(c)  
  
# The equals sign in Python and many programming languages is assignment, not equality  
# c stores the value of a + b, not the equation itself.  
  
5
```

```
In [17]: # However if we run the cell with c = a + b, then the value of c is updated  
c
```

Out[17]: 5

Types of variables

- <http://phy224.ca/03-types-conversion/index.html> (<http://phy224.ca/03-types-conversion/index.html>)

Numbers

- <https://docs.python.org/3.6/tutorial/introduction.html#numbers>
(<https://docs.python.org/3.6/tutorial/introduction.html#numbers>)

```
In [18]: # Integer
a = 5

# Float - corresponds to double precision in languages like C
pi = 3.1415

# Complex
z = 1.5 + 0.2j # Note that j is what makes this into a complex number
z = complex(1.5, 0.2)
```

Strings

- <https://docs.python.org/3.6/tutorial/introduction.html#strings>
(<https://docs.python.org/3.6/tutorial/introduction.html#strings>)

```
In [19]: # Strings can be created with different types of quotes
first_name = 'Nicholas'
last_name = "Lee-Hone"

# Multiline strings are created with triple single-quotes or triple double-quotes
address = """8888 University Drive
Burnaby, BC V5A 1S6"""
address = '''8888 University Drive
Burnaby, BC V5A 1S6'''
```

```
In [20]: # Remember, variables do not have fixed type, unlike languages like C and Java
a = 10
print(a)
a = 'hello'
print(a)
a = [1, 2, 3]
print(a)
a = complex(1.5, 0.2)
print(a)

10
hello
[1, 2, 3]
(1.5+0.2j)
```

Lists and tuples

- <https://docs.python.org/3.6/tutorial/introduction.html#lists>
(<https://docs.python.org/3.6/tutorial/introduction.html#lists>)
- <https://docs.python.org/3.6/tutorial/datastructures.html#tuples-and-sequences>
(<https://docs.python.org/3.6/tutorial/datastructures.html#tuples-and-sequences>)
- <http://phy224.ca/11-lists/index.html> (<http://phy224.ca/11-lists/index.html>)

```
In [21]: # Lists can contain ANY other variable type
# Lists can be modified after creation
a_list = [1, 2, 'A string', [4, 5, 6], [[1, 2], [3, 4]]]
print(a_list)
print(len(a_list)) # len(a_list) gets the number of elements in the
                  # collection

# Tuples are like lists but you cannot edit their contents after crea
tion
a_tuple = (1, 2, 'A string', [4, 5, 6], [[1, 2], [3, 4]])
print(a_tuple)
print(len(a_tuple))

[1, 2, 'A string', [4, 5, 6], [[1, 2], [3, 4]]]
5
(1, 2, 'A string', [4, 5, 6], [[1, 2], [3, 4]])
5
```

Dictionaries and sets

- <https://docs.python.org/3.6/tutorial/datastructures.html#dictionaries>
(<https://docs.python.org/3.6/tutorial/datastructures.html#dictionaries>)
- <https://docs.python.org/3.6/tutorial/datastructures.html#sets>
(<https://docs.python.org/3.6/tutorial/datastructures.html#sets>)

```
In [22]: # Dictionaries use key:value pairs to store information
# Keys can be strings or other simple values
# Dictionaries can have new entries added after creation and the values can be modified
student = {'first_name': 'ABC',
           'last_name': 'DEF',
           'scores': [10, 20, 30],
           'student_id': 123,
           42: 'Integer...',
           3.14: 'pi',
           2.0 + 1.j: 'complex'
          }
print(student)
print(len(student))

# Sets contain only one copy of a value
a_set = {0, 1, 1, 1, 0, 'a', 'b', 'a'}
print(a_set)
print(len(a_set))

{'first_name': 'ABC', 'last_name': 'DEF', 'scores': [10, 20, 30], 'student_id': 123, 42: 'Integer...', 3.14: 'pi', (2+1j): 'complex'}
7
{0, 1, 'b', 'a'}
4
```

Swapping variables

```
In [23]: # In most programming languages we would need a temporary variable
a = 1
b = 2
print(a, b)
tmp = a
a = b
b = tmp
print(a, b)

1 2
2 1
```

```
In [24]: # A comma separated list creates a tuple automatically
a = 1, 2, 3, 4
a
```

Out[24]: (1, 2, 3, 4)

A comma separated list with no parentheses is interpreted as a tuple. This is not commonly used as it takes extra mental effort to remember that it is a tuple that is created, not a list.

```
In [25]: # Assigning directly into a tuple of variables
(a, b, c, d) = (10, 20, 30, 40)
print(c)
```

30

However, we can also assign directly into a tuple of variables, meaning that we can assign multiple values at the same time.

```
In [26]: # We can use these two things together to do a variable swap without
         # a temporary variable
a = 1
b = 2
print(a, b)
a, b = b, a # Alternatively (a, b) = (b, a)
print(a, b)
```

1 2
2 1

Indexing collections

- <https://docs.python.org/3.6/tutorial/introduction.html#lists>
(<https://docs.python.org/3.6/tutorial/introduction.html#lists>)

```
In [27]: # Use square brackets to access elements in a collection type
         # Python is zero-indexed so the first element in a list or tuple is at
         # index 0, and the last index
         # of a length N list is at index N-1
print(a_list)
print(a_list[0])
print(a_list[len(a_list) - 1])
# There is an easier way to access elements at the end of a list
print(a_list[-1])
```

[1, 2, 'A string', [4, 5, 6], [[1, 2], [3, 4]]]
1
[[1, 2], [3, 4]]
[[1, 2], [3, 4]]

```
In [28]: # Tuples work in the same way
print(a_tuple)
print(a_tuple[2])
```

(1, 2, 'A string', [4, 5, 6], [[1, 2], [3, 4]])
A string

```
In [29]: # Getting nested elements requires reading from left to right
print(a_list)
print(a_list[4])
print(a_list[4][1])
print(a_list[4][1][0])
```

```
[1, 2, 'A string', [4, 5, 6], [[1, 2], [3, 4]]]
[[1, 2], [3, 4]]
[3, 4]
3
```

```
In [30]: # Getting an element from a dictionary is similar, but the key is now
         use in the square brackets
print(student)
print(student['scores'])
print(student['scores'][1])
```

```
{'first_name': 'ABC', 'last_name': 'DEF', 'scores': [10, 20, 30], 'st
udent_id': 123, 42: 'Integer...', 3.14: 'pi', (2+1j): 'complex'}
[10, 20, 30]
20
```

```
In [31]: # Lists and tuples can be sliced. I.e. we can get more than one value
         at a time
print(a_tuple)
print(a_tuple[2:])
print(a_tuple[:-2])
print(a_tuple[1:-1])
print(a_tuple[:2])
# Full syntax for slicing is [start:stop:step]
```

```
(1, 2, 'A string', [4, 5, 6], [[1, 2], [3, 4]])
('A string', [4, 5, 6], [[1, 2], [3, 4]])
(1, 2, 'A string')
(2, 'A string', [4, 5, 6])
(1, 'A string', [[1, 2], [3, 4]])
```

```
In [32]: # Note that a string is also a collection of letters, so we can acces
         s the elements of a string
         # in the same way
print(first_name)
print(first_name[3])
print(first_name[1:3])
```

```
Nicholas
h
ios
```

```
In [33]: # Can we get a value from a set?
a_set[0]
```

```
-----
-
TypeError                                Traceback (most recent call last)
<ipython-input-33-e10430808d94> in <module>
      1 # Can we get a value from a set?
----> 2 a_set[0]

TypeError: 'set' object does not support indexing
```

For more information on errors see the documentation: <https://docs.python.org/3.6/tutorial/errors.html>
(<https://docs.python.org/3.6/tutorial/errors.html>)

Error traces are meant to be read from bottom to top.

The error name and description is at the bottom.

The line on which the error occurred is at the top of the trace.

```
In [34]: # Adding, modifying, and removing elements from a list
my_list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(my_list)

# We can add elements to the end using the .append() method
my_list.append(11)
print(my_list)

# Add elements at a particular position using the .insert(position, value) method
my_list.insert(0, 'first!')
print(my_list)

# Delete elements using the del keyword
del my_list[4]
print(my_list)

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
['first!', 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
['first!', 0, 1, 2, 4, 5, 6, 7, 8, 9, 10, 11]
```



```
In [35]: print(my_list)

# Modify elements at some index
my_list[4] = 'Test'
print(my_list)

# Combine these with slices
del my_list[-4:]
print(my_list)

my_list[:3] = ['A', 'B', 'C']
print(my_list)

['first!', 0, 1, 2, 4, 5, 6, 7, 8, 9, 10, 11]
['first!', 0, 1, 2, 'Test', 5, 6, 7, 8, 9, 10, 11]
['first!', 0, 1, 2, 'Test', 5, 6, 7]
['A', 'B', 'C', 2, 'Test', 5, 6, 7]
```

Ranges

<https://docs.python.org/3.6/tutorial/controlflow.html#the-range-function>

(<https://docs.python.org/3.6/tutorial/controlflow.html#the-range-function>)

```
In [36]: # Does not create a list. Just stores the endpoints, so it is computationally efficient
range(10)
```

Out[36]: range(0, 10)

```
In [37]: # A range does act like a list though, and we can index it
range(10, 100)[5]
```

Out[37]: 15

```
In [38]: # We can also turn it into a list
# Note that the max value is left out
print(list(range(3, 20)))
print()

# This is because we normally use ranges to iterate over a list or tuple
print(a_list)
print(len(a_list))
print(range(len(a_list)))
print(list(range(len(a_list))))
```

[3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

[1, 2, 'A string', [4, 5, 6], [[1, 2], [3, 4]]]

5

range(0, 5)

[0, 1, 2, 3, 4]

Ranges are typically used when looping over a list or a tuple.

We will see later how this is used, but for now it is important to note that the values produced by `range(len(a_list))` match the indices needed to access the elements in that list.

Conditional evaluation

- <https://docs.python.org/3.6/tutorial/controlflow.html#if-statements>
(<https://docs.python.org/3.6/tutorial/controlflow.html#if-statements>)
- <http://phy224.ca/17-conditionals/index.html> (<http://phy224.ca/17-conditionals/index.html>)

```
In [39]: # Conditional evaluation. I.e. if statements
# Syntax is:
# if (condition):
#     tab-indented code

if True:
    print('It is true')

# Note no output
if False:
    print('It is false')
```

It is true

```
In [40]: a = 4
if a < 5:
    print('Less than 5')
else:
    print('Greater than or equal to 5')
```

Less than 5

```
In [41]: # There can be as many 'elif' statements in between
# Remember that the conditions are evaluated in order from top to bot
tom.
# If one of the conditions is true, the rest of the code below won't
be evaluated
a = 12
if a < 5:
    print('Less than 5')
elif a >= 10:
    print('Greater than or equal to 10')
elif a >= 5:
    print('Greater than or equal to 5')
    # Note that if a is e.g. 6, then it prints the line above,
    # but if a is e.g. 12 it prints 'Greater than or equal to 10'
    # despite this case being True
else:
    print('In between')
```

Greater than or equal to 10

```
In [42]: # Remember that the conditions are evaluated in order from top to bot
tom.
# If one of the conditions is true, the rest of the code below won't
be evaluated
a = 12
if a < 5:
    print('Less than 5')
if a >= 10:
    print('Greater than or equal to 10')
if a >= 5:
    print('Greater than or equal to 5')
else:
    print('In between')
```

Greater than or equal to 10
Greater than or equal to 5

Note that in the second case there are two lines of output. Each if statement actually represents a conditional block and each block is evaluated independently of all the others.

```
In [43]: # Create more complex logic with the 'and'/'or' keywords
a = 9
if a < 5 or a > 10:
    print('Out of bounds')
elif a == 6:
    # Note the use of double equals here.
    # Double equals sign is to check for equality, single equals sign
    # is to assign to a variable
    print('Is 6')
elif a != 7 and a != 8:
    print('Not 7 or 8')
else:
    print('Other case')
```

Not 7 or 8

```
In [44]: # Conditionals can be nested
a = 4
b = 5
if a < 5:
    if b > 0:
        print('Positive b')
    else:
        print('Negative b')
elif a < 10:
    print("This is true, but won't run")
else:
    print("Some other case")
```

Positive b

Repeating code multiple times (looping)

- <https://docs.python.org/3.6/tutorial/controlflow.html#for-statements>
(<https://docs.python.org/3.6/tutorial/controlflow.html#for-statements>)
- <http://phy224.ca/12-for-loops/index.html> (<http://phy224.ca/12-for-loops/index.html>)
- <http://phy224.ca/13-looping-data-sets/index.html> (<http://phy224.ca/13-looping-data-sets/index.html>)

There are two types of loops in Python: for loops and while loops.

A for loop is used when you know how many iterations you want to run.

A while loop is used when you don't know ahead of time how many iterations to run.

```
In [45]: # We can iterate through the elements accessing each one individually by index  
print(a_list)  
for i in range(len(a_list)):  
    print(i, ': ', a_list[i])
```

```
[1, 2, 'A string', [4, 5, 6], [[1, 2], [3, 4]]]  
0 : 1  
1 : 2  
2 : A string  
3 : [4, 5, 6]  
4 : [[1, 2], [3, 4]]
```

```
In [46]: # Or we can just iterate over the elements directly - this is the preferred version  
print(a_list)  
for list_element in a_list:  
    print(list_element)
```

```
[1, 2, 'A string', [4, 5, 6], [[1, 2], [3, 4]]]  
1  
2  
A string  
[4, 5, 6]  
[[1, 2], [3, 4]]
```

```
In [47]: for list_element in a_list:  
        list_element = 10  
print(a_list)  
# Note that there has been no change because we were modifying the label location, not the value in the list
```

```
[1, 2, 'A string', [4, 5, 6], [[1, 2], [3, 4]]]
```

```
In [48]: for i in range(len(a_list)):  
        a_list[i] = 10  
print(a_list)
```

```
[10, 10, 10, 10, 10]
```

```
In [49]: # What if we need to both iterate over the list and modify it?
a_list = list(a_tuple) # Create a copy of a_tuple that is actually a
                        # list
print(a_list)

for i, val in enumerate(a_list):
    a_list[i] = val*3
    # Equivalent to a_list[i] = a_list[i]*3
print(a_list)

# Note that strings can be multiplied by integers, and so can lists a
# nd tuples!
# However dictionaries and sets cannot be multiplied by integers

[1, 2, 'A string', [4, 5, 6], [[1, 2], [3, 4]]]
[3, 6, 'A stringA stringA string', [4, 5, 6, 4, 5, 6, 4, 5, 6], [[1,
2], [3, 4], [1, 2], [3, 4], [1, 2], [3, 4]]]
```

```
In [50]: # While loops
i = 0
while i < 10:
    print(i)
    i += 1 # Shortened form of i = i + 1
    # REALLY important to have i increment otherwise infinite loop

0
1
2
3
4
5
6
7
8
9
```

Break and continue

<https://docs.python.org/3.6/tutorial/controlflow.html#break-and-continue-statements-and-else-clauses-on-loops>
[\(https://docs.python.org/3.6/tutorial/controlflow.html#break-and-continue-statements-and-else-clauses-on-loops\)](https://docs.python.org/3.6/tutorial/controlflow.html#break-and-continue-statements-and-else-clauses-on-loops)

```
In [51]: # Break out of a while loop
i = 0
while True:
    print(i)
    i += 1
    if i == 10:
        break # Stops the while loop. Again, REALLY important to have this or we have an infinite loop
```

```
0
1
2
3
4
5
6
7
8
9
```

```
In [ ]: # DO NOT RUN THIS CELL! If you do, press the stop button in the toolbar (black square).

# Skip iterations of a loop (we don't want to print when i is less than 4)
i = 0
while True:
    if i < 4:
        continue # Skips the rest of the code inside the loop.
        # This is useful when you have very complex code that comes later and you want to skip it

    print(i)
    i += 1
    if i == 10:
        break
```

```
In [53]: # Skip iterations of a loop (we don't want to print when i is less than 4)
i = -1 # Start one before because we want the first instance of i in the loop to be zero
while True:
    i += 1
    if i < 4:
        continue # Skips the rest of the code inside the loop

    print(i)
    if i == 10:
        break
```

```
4
5
6
7
8
9
10
```

Functions

- <https://docs.python.org/3.6/tutorial/controlflow.html#defining-functions>
(<https://docs.python.org/3.6/tutorial/controlflow.html#defining-functions>)
- <https://docs.python.org/3.6/tutorial/controlflow.html#more-on-defining-functions>
(<https://docs.python.org/3.6/tutorial/controlflow.html#more-on-defining-functions>)
- <http://phy224.ca/04-built-in/index.html> (<http://phy224.ca/04-built-in/index.html>)
- <http://phy224.ca/14-writing-functions/index.html> (<http://phy224.ca/14-writing-functions/index.html>)
- <http://phy224.ca/15-scope/index.html> (<http://phy224.ca/15-scope/index.html>)

```
In [54]: # Functions allow you to write reusable pieces of code
# The parameter names (in this case, a and b) shadow the variable names that we have already declared
# Think of a and b as being placeholders for values that will be inserted later
def add(a, b):
    return a + b

c = add(4, 100)
print(c)
print(add(2, 3))
print(add(6, 8))
```

```
104
5
14
```



```
In [55]: # Functions can use other functions
def add_10(a):
    return add(10, a)

print(add_10(2))
```

12

```
In [56]: # Functions can take other functions as parameters
def math(func, a, b):
    print("Calculating the results of")
    print(func)
    print('applied to')
    print(a, b)
    print('is')
    return func(a, b)
```

```
In [57]: math(add, 2, 3)
```

Calculating the results of
<function add at 0x7f8024310e18>
applied to
2 3
is

Out[57]: 5

```
In [58]: # Let's try another function
def divide_by_b_squared(a, b):
    return a / b**2

print(math(divide_by_b_squared, 3, 4))
```

Calculating the results of
<function divide_by_b_squared at 0x7f8024bd9d08>
applied to
3 4
is
0.1875

This becomes extremely powerful once you want to create things like integration (quadrature) routines because you can pass the function you want to integrate to the integration function. No need to write an integrator for each function you can imagine.

This was a very condensed overview of the features that are most relevant for the upcoming lectures.

Not covered here, but will be in later lectures:

- Loading a package
- Keyword parameters and default parameter values
- Variable packing and unpacking
- List comprehension
- Lambda functions
- Loading and saving files

Part of Python core, but will not be covered as they aren't as useful in a teaching environment:

- Recursion
- Generators
- Decorators
- Classes and object oriented programming
- Creating your own modules and packages
- Speeding up python code with Cython and numba

In []: