

Welcome to an introduction to Python

G. Benoît

Before we begin ...

Required texts:

- Cannon, J. (2016). *Python succinctly*.
- Severance, C. (2016). *Python for everyone: exploring data in python 3*. [<https://www.isi.edu/~pedro/Teaching/INF510-Spring17/Materials/book-python3.pdf>]

There are, imho, a few basics that *everyone* who uses computers for their job need to know, so the purpose of our introduction is to expose the student to an intersection of several computing fundamentals. These include: operating systems, encoding, stdin, stdout, stderr, and computing architectures, and how ultimately we apply these skills in all aspects of information systems & services. Our first fusion is learning about scripting languages, using python. The skills used in python transfer easily to other computer languages, too, such as [php](#), [JavaScript](#) and R. With python under you belt, you'll be able to learn more on your own and explore other computer languages as you need them for fun and on your job.

Required Software:

Python: The latest version of Python, as of July 3, 2019, is version 3.7.5. For the course, you should have installed any version of Python 3.0 or later. Commands used in earlier versions may not be supported by Python 3. Please note that if you're using a Mac, you *may* have to interact with the terminal window by using `python3` as opposed to just `python` as you'll see in the course and elsewhere.

Text editor: You need a text editor - not a word processing program nor TextEdit - to create scripts. There are several options out there; I like BBEdit but others use Atom, NotePad+. Some student may have experience with IDLE and Jupyter Notebooks. IDLE is a common interactive interface scripting tool; Jupyter Notebooks are an interactive, .json-based development environment. IDLE is usually included in the standard distribution of Python; Notebook needs to be downloaded and installed independently. Some students download the entire Anaconda suite of data science tools that includes Jupyter notebook, R, and other tools. [Search online for the latest versions.]

Canvas: Our primary engagement for teaching and communicating is through Canvas [<https://ischool.sjsu.edu/canvas>]. And there you're *encouraged strongly* to ask questions, answer others' questions, get to know each other, explore ... Secondly, you're invited to send me questions and I'll do my best to answer. If you don't receive a reply within a few hours, let me know - it is likely that your note went accidentally to spam and so never delivered or accidentally I missed it.



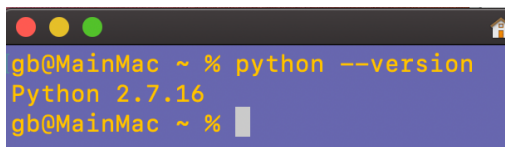
Using python

There are a couple of ways to interact with python, the scripting language of choice for data science.

1. Write a script using a **text editor**, save the file with the extension `.py`, and then execute the script from a terminal window by starting the python interpreter. [Remember, the `%` or `$` symbols used in our examples are the terminal window prompt; don't type it, e.g.:
 - a. `$ python myscript.py`
2. A very popular training tool you should explore is the interactive toolkit [Jupyter Notebook](#), usually part of the suite of [Anaconda](#) portal. A "notebook" is interactive: you type the code and the run it. The notebook executes the code and shows any errors.
3. [Spyder](#) and other IDEs (Integrated Development Environments).
4. Python is a scripting language, meaning you write scripts (the programs) and they're executed by an "interpreter." All scripting languages include libraries of code that do certain tasks more efficiently. These **modules** include *matplotlib* and *seaborn* (and others) for visualization and analysis, and *pandas* and *numpy* for analysis and data structures that work well with `.xml`, `.json`, tab-delimited, `.csv` files.

Downloading and installing python

1. First check whether python is already installed on your computer. On Mac/Unix/Linux OS computers, python is already installed. Access the command-line/terminal window:
 - a. Linux: Ctrl-Alt-T, Ctrl-Alt-F2
 - b. Windows: Win+R > type `powershell` > Enter/OK
 - i. You may want to read more about Python on Windows on [Python's documentation site](#).
 - c. MacOS: Finder > Applications > Utilities > Terminal
2. Once the terminal window is ready, issue the command to check: `python --version`



```

gb@MainMac ~ % python --version
Python 2.7.16
gb@MainMac ~ %

```

3. There may be multiple versions on your machine: check by typing `python --version` (for versions 2.7.x) and `python3 --version` (for versions 3.x). *Note: depending on your computer setup you may need to replace the command "python" with "python3" throughout the rest of these readings and exercises.*

As a fun little test, let's start a text editor application, such as [BBEdit](#), [Atom](#), NotePad+, and write our first little script. This script you'll run from the terminal window!

Your first script - test whether python is installed:

Wait! Before you create the file, remember that python relies on *indentation* to identify "code blocks." The indentation must be *either* four spaces *or* a tab; never both in the same file. So in the demo below, the first "print" line is intended by tapping the space bar four times.

```

import sys

if not sys.version_info.major == 3 and sys.version_info.minor >= 6:
    print("Python 3.6 or higher is required.")
    print("You are using Python {}.{}.".format(sys.version_info.major, sys.version_info.minor))
    sys.exit(1)

```

When you're done save this file as "versiontest.py" [notice there are no spaces in the file name]. Make sure your current working directory (e.g., "Documents/pythonprojects/") is the same that holds your new .py file.

```
gb@MainMac ~ % cd Documents
gb@MainMac Documents % python3 versiontest.py
```

All computer programming and scripting languages rely on "control of flow" statements, functions and methods, data structures, file reading/writing, and error checking. Of course there is a lot more and each language has its own syntax but the concepts remain strong across all languages. We'll explore each of these now.

Using Python's Command Line

Keeping to our terminal window we'll start the actual python program (or "interpreter") that will interpret our commands and respond immediately. To **exit** python, enter **exit()** and press the return key.

1) Start python by typing **python** and pressing the enter or return key.

```
gb@MainMac Documents % python3
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 16:52:21)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

2) Notice python has its own prompt: the three >>>

Functions and methods are pre-written commands that we use as the basics of the language. Functions are independent, defined blocks of code that perform some task. Functions may accept "arguments" (or parameters), data that we supply for the function to do its job. Functions may "return" a value, the result of the function's work. In this example the `print()` function takes an argument "Hello!", and "prints" the results on the stdout (the monitor).

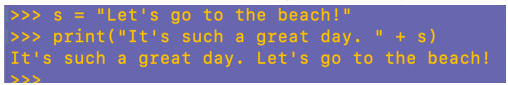
```
>>> print("Hello!")
Hello!
>>> x = "Hi, again!"
>>> print(x)
Hi, again!
>>>
```

We can also **declare variables** and use them. A **variable** is a container with a name that holds a value. E.g., "x" is a variable name as could be `welcome_greeting`. Variables cannot begin with certain symbols (such as period, colon, exclamation point) and cannot contain spaces. Once a variable is "declared" and assigned a value "instantiated" we can use it in code. Here we see "`print()`" function being passed a parameter ("Hello"); next we see a variable declared (x) and passed as a parameter, too. And the value of the parameter is used in the output.

3) **Control-of-Flow and Loops; Variables and more:**

Demo Table 1	
#comments	It's best to include comments in the code so you'll remember what you're doing and others can debug the code. There are 2 kinds: the # and a comment block using <code>"""</code>

if	<pre># an if statement if 5 > 3: print("five is indeed bigger than 3") # an if statement using variables x = 5 y = 3 if x > y: print("and it works with variables")</pre>
if ... elif	<pre># an if-elif statement using variables x = 5 y = 3 if x > y: print("Five is indeed bigger!") elif x == y: print("well, x and y are equal! Weird.")</pre>
if ... else	<pre># an if-else statement using variables x = "Tom" y = "Turkey" if x == y: print("The two variables are equal.") else: print("No, the two variables are not the same.")</pre>
Shortcuts	<pre>x = 82 y = 92 if y > x: print("You scored and A") """ in the above statement notice the syntax. if the value of y is greater than that of x, then print "You score and A." """ print("Grade A") if y > x else print("B") """ notice the syntax about the outcome (the print) appears first, then the comparison - the if statement followed by the else. Get fa- miliar with this notation because it's used in "lambda functions". """</pre>
Combine comparisons with and	<pre>a = 5 b = 10 c = 15 if a > b and a > c: print("both statements are true.") if a > b or a > c: print("at least one of these statements is true.")</pre>
Nested if statements	<p>Sometimes we want to check a condition and see if it is true. If that condition is true, then we might want to do tests on other conditions. For example, "if the student is in class 100" we want to assign grades.</p> <pre>student_in_class = True score = 93 if student_in_class: # same as if student_in_class == True: if score > 90: print("Grade A") else if score > 80 and score <= 89: print("Grade B") else if score > 70 and score <= 70: print("Grade C") elif: print("Grade D, sorry.")</pre>

Pass statement	<p>There are times when you're coding and you kind of know what you want to include but you're not finished. The <code>pass</code> statement allows us to use the code even if we're not ready to finish the code and not get an error.</p> <pre>student_Name = "Tom" is_enrolled = True if is_enrolled: pass</pre>
# types of variables built-into python	<p>Depending on the kind of data we have we use different kinds of variables to hold those data. Here's the basics.</p> <p>integer counting numbers, e.g., 0,2,3,6,333,142396 floating point 3.141458372 char 'a', 'b' string "Hi, folk" can use single quotes, too, 'Hey!'</p> <p>str for text, e.g., x = "Smith" int, float, complex for numbers, e.g., i = 3.1432 sequence types list, tuple, range mapping types dict boolean types bool</p> <p>[there are others but we'll not use them now.] Notice that strings are wrapped in quotes. Numbers are not. This is how python determines the data type.</p>
Using variables. Variables can be "global" or "local." A global var can be used anywhere in the code. A local variable is used only in the code block where it is declared.	<pre>""" Notice that the quotes can be double or single when wrapping a string. That's how we can use the ' in the word let's. If the quotes are unbalanced then the code will fail. """ s = "Let's go to the beach!" print("It's such a great day " + s)</pre> 

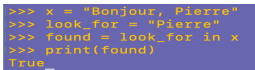
Test yourself on the contents of Demo Table 1. When you understand it all, proceed to the next table(s).

Demo Table 2	Knowing the value of a variable and what parts of the code can change the value is awfully important! Let's practice controlling the "visibility" of the variable by defining some functions and testing. Then we'll "promote" the variable to global as a test.
defining a function	<pre>""" functions require a "keyword", def, to define a function's code block. Then any variable declared in that block is visible only to that block of code. Imagine if you have two variables called x ... """</pre>

<p>the keyword <code>def</code> for defining functions</p>	<pre> x = "fun" # notice this var is not part of a function; it is global def whatToDo(): x = "Go to the beach" print("What to do? Something like " + x) # to end our function press the return key twice. whatToDo() # notice that functions are defined but not used # until we "call" them into action print("Let's do something " + x) </pre>  <pre> >>> s = "Let's go to the beach!" >>> print("It's such a great day. " + s) It's such a great day. Let's go to the beach! >>> x = "fun" >>> def whatToDo(): ... x = "Go to the beach" ... print("What to do? Something like " + x) ... >>> whatToDo() What to do? Something like Go to the beach >>> print("Let's do something " + x) Let's do something fun >>> </pre>
<p>Data Type samples:</p> <p>(there is also complex, set, frozenset, bytes, bytearray, and memoryview types but we'll not use these in our course.)</p>	<pre> x = "Hello, Dave" string type (str) x = 20 integer (int) x = -39.1 floating point (float) x = ["cat", "dog", "cow"] a list - notice the use of []. Lists are great for using lots of variables. x = ("cat", "dog", "cow") a tuple - notice the () x = range(10) a range, extremely useful x = { "name": "Ricky", "age", "27"} a dictionary (dict) - invaluable especially with .json files x = True boolean (True or False) </pre>
<p>Converting between data types</p>	<p>Say we have a variable "x" that refers to an integer, 10. We want to convert that integer to a floating point value. Use the "constructor": e.g.,</p> <pre> x = 10 print(x) 10 x = float(10) print(x) 10.0 </pre> <p># notice the output</p> <p># always good to confirm. Try it.</p>

Practice creating lots of variables and printing them. "Cast" (or convert) the variable between different types and see if works and if not what the error messages look like. It's *vital* that you become accustomed to error message, not stressing over them (grin) and learning how to read the error message for debugging your code.

<p>Demo Table 3</p> <p>“Arrays” - python like most language stores “strings” as an array of unicode characters.</p>	<pre> x = "Buenas días" print(x) print(x[0]) print(x[1]) # this is a string # and the entire sting prints # the [] refers to the "index" or the # location of some value at the # used # the letter "B" prints (location = 0) # returns "u" </pre>
<p>knowing where data are being stored is really important. And there are several ways to accessing the data - by position or “index” is common.</p>	<pre> """ Slicing data - we select the start number, the last number, and the number of characters to take at a time """ print(x[0:5:1]) print(x[0::]) print(x[0:len(x):]) print(len(x)) # prints "Buena" # prints "Buenas días" - since we add no # values for last and number at a time # python assumes we want all the data. # notice len(x) - this means "get the # length of the word and store value here # then complete the splice command. # let's check ... 11 </pre>

Demo Table 4	
	<p>Strings There is a lot of string processing used in information science and informatics. In this demo table we look at some of the most important. It's always important to get to know your data - make no assumptions - and check. Sometimes even an extra space at the end of the word might wreck our code. And when comparing values (e.g., "Hello" ≠ "HELLO") we might want to convert data to all uppercase or all lowercase and so on. Let's try each of these in your terminal window. Using the same string in all examples: x = "Bonjour, Pierre". We send the output of our string commands to the print method so we can confirm.</p>
upper case lower case strip off whitespaces	<pre>x = "Bonjour, Pierre" print(x.upper()) print(x.lower()) print(x.strip())</pre>
checking strings is a substring found?  <pre>>>> x = "Bonjour, Pierre" >>> look_for = "Pierre" >>> found = look_for in x >>> print(found) True</pre>	<pre>""" sometimes we check whether data are found in a string. If we we might want to replace some characters. Or we might want to break up a string into "tokens" based on some delimiter. This is often the first step in data cleaning and parsing text files for full-text retrieval algorithms. """ look_for = "Pierre" # use the "in" or "not in" keywords: found = look_for in x # is "Pierre" found in the string x? print(found) """ now you try - using the "not in" keyword."""</pre>
breaking up strings splitting replacing breaking up strings with the \t tab	<pre>""" often need to break up strings based on some delimiter, such as the tab (or \t) or a comma. """ print(x.split(",")) # returns ['Bonjour', 'Pierre'] """ now you try - using the "not in" keyword. """ s = "Bontour, Pierre" # the "t" is an error; let's fix that. s = s.replace("t", "j") # the first instance of t is now j print(s) """ here the data are separated by an invisible tab (\t). I inserted the \t to demo its location. NOTE: this will not work on the terminal line - but it's something we use all the time IN the .py source file.""" s = "Tom Jones\t50,000\tRoom 32\t333-1234" """ because it is SO common to export data from a spreadsheet into a "tab-delimited file", we need to know how to extract our data """ print(s.split("\t")) # what do you see?</pre>

<p>About “escape characters” - tab, newline, carriage return</p> <p><code>\t</code></p> <p><code>\n</code></p> <p><code>\r\n</code></p> <p>and apostrophes/quotes</p> <p><code>\'</code> for apostrophe</p> <p><code>\"</code> for quotes</p>	<p>An “escape” character tells the computer to “escape” from its usual processing and treat the following character in a special way. The most common for beginners is “tab”, “new line”, and “carriage return.”</p> <p>Tab is indicated by “<code>\t</code>”</p> <p>New line (as if you pressed the return key) is “<code>\n</code>”</p> <p>Because quotes <i>must</i> be balanced, if we have a string of data wrapped in “ ” and then a “ inside the string, we must “escape” the quote: e.g.,</p> <pre>s = "We've won the "Nobel Prize" for physics!" must become s = "We've won the \"Nobel Prize\" for physics!"</pre> <p>For WINDOWS only there’s often a hassle because Windows used to use “<code>\r\n</code>” (carriage return + new line) and that causes a problem interpreting the combination. Programming languages usually will convert the “<code>\r\n</code>” to just “<code>\n</code>”. BUT you cannot be sure about this when reading a data file! Just fyi.</p>
--	--

Practice creating strings and manipulating them.

Demo Table 5	Booleans
With booleans we can test the data, a function, and write code using our control-of-flow statements.	<pre>x = 10 print(x == 10) # the == means “is equivalent to” # so read the statement as “print the result # true if x is equivalent to 10, print false” """ try using the other operands, too: > == < >= <= """</pre>
Using in code	<pre>x = 1000 y = 1032 if x < y: print(x, “ is less than ”, y) else: print(y, “ is less than ”, x)</pre> <p>""" Notice the print statement. Python let’s us send a many arguments, separated by a comma, and the string appears correct on screen """</p>

<p>Using in a function as a return value</p> <p>Here, notice at the bottom of the code snippet, x and y are assigned values. Then we pass those values to the function which_is_greater. That function springs into action and sends back to the part of the code that called it the results - the "s"</p>	<p>When we reuse the same calculations or commands a lot it is useful to store them in a function. And since functions can "return" the result of its work and store that result in a variable, it's extremely useful to combine functions and booleans. Here say we're often asking if two variables are greater or less than each other. So let's write a function:</p> <pre>def which_is_greater(x, y): if x < y: s = print(x, " is less than ", y) else: s = print(y, " is less than ", x) return s print("lets compare two variables") x = 300 y = 400 which_is_greater(x, y)</pre>
---	--

So far we have the basics; now let's build upon this foundation.

Demo Table 6	for ... and while loops
<p>for ...</p> <p>while loops</p>	<p>Usually we want to keep something running until something causes it to stop. For example, when we write a program, we want to keep it running until the end-user chooses to quit. Equally, we might already know when to quit something when we have a limited number of items to check, for example, let's say we have five cards. We could say "while we have cards in our hand, lay them down on the table one by one." Often we have a variable just for counting. So we need to count which card we're using and compare that number to the total number of cards. The first time we lay down a card, the count is 1, and then 2, and so on until we reach the end of the cards.</p> <pre>count = 1 number_of_cards = 5 while count < number_of_cards: print("Card # ", i) count += 1 # if we don't increment the count, the loop continues forever</pre>
<p>break statement stops the loop even if the while condition is true</p>	<pre>i = 1 while i < 6: print(i) if i == 3: break i += 1</pre>
<p>continue continues the next iteration, in this case, even if i is 3.</p>	<pre>i = 0 while i < 6: i += 1 if i == 3: continue print(i)</pre>

while and else together! Prints a message once the condition is false.	<pre> i = 1 while i < 6: print(i) i += 1 else: print("i is no longer less than 6") </pre>
--	--

The fundamentals we've seen and hands-on practice should provide a reasonable basis for benefiting from the online text books. In this example, we want to create a menu for end-users. The program runs in a terminal window until the end-user chooses to exit. There are four options: 1, 2, 3, and q. Let's say 1 = "English", 2 = "Español", 3 = "Français", and 0 = quit.

Demo screen:

Hi. In what language would you like your welcome?

1 = English
 2 = Español
 3 = Français
 0 = Quit

Enter your choice:

Demo code for the above:

```

print("-" * 60)                                # means print - for 60 times
print("Hi. In what language would you like your welcome?\n")
print("1 = English")
print("2 = Español")
print("3 = Français")
print("0 = Quit")
useroption = (int)input("Enter your choice:")
print("-" * 60)

# now let's check the values of our choices. We accept ONLY 1, 2, 3, and the letter "q"
if (useroption in range(0,4)):                  # if the user choice is legit (0, 1, 2, 3)
    # nested if example
    if useroption == 1:
        print("Welcome to our class")
    else if useroption == 2:
        print("¡Bienvenido a nuestra clase!")
    elif:
        print("Bienvenue à notre cours.")
else:
    print("Sorry, only values 1, 2, 3, and 0 are okay.")

```

The above script will run only one time. Let's keep it running until the end-user chooses to exit by applying the while statement. Notice below the use of (int). This means whatever the end-user enters should be converted to an integer. Notice, tho, if the person enters "a", then it mayn't work.

```
print("-" * 60)                                # means print - for 60 times
print("Hi. In what language would you like your welcome?\n")
print("1 = English")
print("2 = Español")
print("3 = Français")
print("0 = Quit")
useroption = (int)input("Enter your choice:")
print("-" * 60)

"""
Here we apply a "boolean flag". That means we set up a condition to be True until the end-
user enters some data that causes the boolean flag to become False. Our flag here is "is-
Done"
"""
isDone = False                                # keep going until "isDone" is true

while !isDone                                # the isDone is "false", using the ! reverse it to true
    # nested if example
    if useroption == 1:
        print("Welcome to our class")
    else if useroption == 2:
        print("¡Bienvenido a nuestra clase!")
    else if useroption == 3:
        print("Bienvenue à notre cours!")
    else if useroption == 0:
        print("Thanks for visiting")
        isDone = True
    elif:
        print("Sorry only 0, 1, 2, 3 values are legit.")
```

Compare the two approaches. Both are fine. In real practice we try to avoid too many nested if statements and combinations of for, while and if all together. The time it takes to process the code and data can increase exponentially. The measurement of coding algorithm design is called Big O. Big O is a measure, you'll recall of the "worst case scenario" of the algorithm. For our purposes, we don't have to worry about it. But should you continue with python and particularly BigData/Data Science, you should read more about this.

Operators	<p>here's a fuller list of the operands:</p> <p>+ addition e.g., <code>x + y</code></p> <p>- subtraction <code>x - y</code></p> <p>* multiplication e.g., <code>5 * 2</code></p> <p>/ division</p> <p>% modulus (the remainder of 2 numbers, e.g., <code>30 % 2 = 0</code>)</p> <p>** exponent, e.g., <code>5²</code> is written <code>5**2</code></p>
Assignment operators	<p>= <code>x = 5</code> assign a value of 5 to the variable <code>x</code></p> <p>+= increment 1, e.g., <code>x += 3</code> is the same as <code>x = x + 3</code></p> <p>-= decrement</p> <p>/= divide, e.g., <code>x /= 3</code> is the same as <code>x = x / 3</code></p>
Comparison operators	<p>== equal to, e.g., <code>x == 5</code> (returns true or false)</p> <p>!= not equal to <code>x != y</code></p> <p>> greater than <code>x > y</code>, e.g., <code>5 > 3</code></p> <p>< less than <code>x < y</code></p> <p>>= greater than or equal to</p> <p><= less than or equal to</p>
Logical operators	<p>and returns True if both statements are true, if <code>x < 5</code> and <code>y > 3</code></p> <p>or returns True if <i>one</i> of the statements is true, <code>x < 5</code> or <code>x < 4</code></p> <p>not reverses the result; false if the result is true, e.g., if not(<code>x < 5</code> and <code>y > 3</code>):</p>
Identity operators	<p>is returns true if both variables are the same object <code>x is y</code></p> <p>is not returns true if both variables are <i>not</i> the same object "same object" means the same memory location</p>
membership operations	<p>in returns True if a sequence with the specified value is present e.g., for <code>x in range(0,5)</code>: means if the value of "x" is not between 0 and 5, then the answer is False.</p> <p>not in returns true if a sequence with the specified value is not present in the object</p>

Continue with part 2.

Below there's a little review sheet for the fundamentals of Python.

Python Review Help Sheet - this is an optional review just fyi guide.

PYTHON 3.x HELP SHEET

Basics

#		x = 5 #int	print(x + y)	x = 1	y = 2.8
# comments	""" multiline """	y = "String"	concatenate Strings or add numbers	int	float
z = 1j	type(varname)	x = int(y)	<-- ex of cast: str(y), float(y), int(y)		

Strings

x = "hello"		a.string()	a.upper()	a = "hello, Tom"
print(x[1])	print(x[2:5])	len(a)	a.replace("e", "f")	a.split(",")
prints "e"	pos 2 to 5 (not including 5)	a.lower()	x = input() # (command line)	

Operators

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

Operator	Description	Example
is	Returns true if both variables are the same object	x is y
is not	Returns true if both variables are not the same object	x is not y

in	Returns True if a sequence with the specified value is present in the object	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y

&	AND	Sets each bit to 1 if both bits are 1
	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

List/Array Methods: thislist = ["honolulu", "italy", "france"]

append()	clear()	copy()	count()	extend()	index()
insert()	pop()	remove()	reverse()	sort()	print(list[1])

Tuple: unchangeable thistuple = ("apple", "fish", "cat")

append()	clear()	copy()	count()	extend()	index()
insert()	pop()	remove()	reverse()	sort()	print(list[1])

Set: unordered, unindexed thisset = {"honolulu", "italy", "france"}

set() thisset = set(("cat", "dog", "bird"))

add()	update()	add multiple items to set: thisset.update(["a","b","c"])			
len()	pop()	remove()	discard()	clear()	del thisset

Dictionary: unordered, changeable, indexed

```
thisdisc = {"brand": "Ford", "model": "Mustang", "year": 1964}
get value: thisdisc["model"]    thisdict.get("model")
change: thisdict["year"] = 2018
```

loop via keys: for x in thisdict: print(x)					
loop via values: for x in thisdict print(thisdict[x])					
for x in thisdict.values():					
loop thru keys & values: for x, y in thisdict.items():					
len()	del thisdict["model"]	pop("model")	popitem()	del thisdict["model"]	del thisset
clear()	dict(brand="a", model="b", year="c")			get()	copy()
get()	fromkeys()			items()	keys()
	setdefault()			update()	values()

if ... else:

if b > a:	if b > a:	if a > b:	print("A") if a > b else print("B")
print()	print("b")	print("x")	print("A") if a > b else print("B") if a==b else print("B")
elif a==b:	elif a==b:		if a > b and c > a: # both conditions met
	print("eq")		if a > b or a > c:
	else:		
	print("a")		

while and for:

while j < 6:	while j < 5:	while j < 5:	f = ['a', 'b', 'c']	for x in "cat"	for x in f:
print(j)	print(j)	j += 1	for x in fruits:	print(x)	if x == "a":
	if j == 3:	if j == 3:	print(x)		break
	break	continue			
	j += 1	print(j)			

Functions: use def

def myfunction():	def myfunc(name="Tom"):	def myfunc(x):
print("hello")	print("I am "+name)	return 5 * x
create a function	create function with default	return value

Lambda: lambda arguments : expression

x = lambda a : a + 10	x = lambda a, b, : a * b	def myfunc(n):
print(x(5))	print(x(5, 6))	return lambda a : a * n
create a function	lambda that multiplies a, b	

Classes & Objects: class

class MyClass:	p1 = MyClass()	class Person:
x = 5	print(p1.x)	def __init__(self, name, age):
		self.name = name
		self.age = age
		p1 = Person("Lars", 25)
		print(p1.name)
create a class	create object p1, print x	__init__()
class Person:		class Person:
def __init__(myobj, name, age):		def __init__(self, name, age):
myobj.name = name		self.name = name
myobj.age = age		self.age = age
		def myfunc(self):
def myfunc(abc):		print("Hello, " + abc.name)
print("Hello, " + abc.name)		p1 = Person("Lars", 25)
p1 = Person("Ryan", 35)		print(p1.name)
p1.myfunc()		
using object instead of self		use self to refer to class vars
	p1.age = 50	del p1.age
	modify	delete
		del p1
		del object

Iterator v. Iterable: lists, tuples, dictionaries, sets, strings are *iterable*.

mytuple = ("a", "b", "c")	mytuple = ("a", "b", "c")	__iter__()
myit = iter(mytuple)		__next__()
print(next(myit))	for x in mytuple:	
iterate thru	print(x)	__init__() to define, next to loop thru

Module: use def for a function; save it in a .py file. Use import

```
import mymodule
mymodule.greeting("Hello")
```

Variables in a module: if dictionary in a module called "mym",
a = mym.person1["age"]

Dates: import datetime

<code>x = datetime.datetime.now()</code>	<code>print(x.year)</code> <code>print(x.strftime("%A"))</code>	<code>x = datetime.datetime(2020,5,17)</code>
<code>insert()</code>	year name of weekday	creates a date object

strftime: formatting time:

%a	%A	%w	%d	%b	%B
Wed	Wednesday	3	31	Dec	December
%m	%y	%Y	%H	%I	%p
12	18	2018	17	05	PM
%M	%S	%f	%z	%Z	%j
41 (min)	08 (sec)	microsecond	UTF offset	eg EST	365 day
%U	%W	%c	%x	%X	%%

json: import json. can convert json to python:

```
import json
x = { "name": "John", "age": 30, "city": "Boston" }
y = json.loads(x)
print(y["age"])
```

Convert from python to json

```
import json
x = { "name": "Ricky", "age": 25, "city": "Boston" }
y = json.dumps(x)
print(y)
```

<code>json.dumps(x, indent=4)</code>	<code>json.dumps(x, indent=4, separators=(".", ","))</code>	<code>json.dumps(x, indent=4, sort_keys=True)</code>
save file with indents	defining separators	sort the data when saving

dict	Object
list	Array
tuple	Array
str	String
int	Number
float	Number
True	true
False	false
None	null

`import json`

demos all of the above conversion types.

```
x = {
    "name": "John",
    "age": 30,
    "married": True,
    "divorced": False,
    "children": ("Ann","Billy"),
    "pets": None,
    "cars": [
        {"model": "BMW 230", "mpg": 27.5},
        {"model": "Ford Edge", "mpg": 24.1}
    ]
}
```

`print(json.dumps(x))`

Try Except: try ... except ... finally

try: <code>print(x)</code> except: <code>print("y")</code>	try: <code>print(x)</code> except NameError: <code>print("y")</code> except: <code>print("z")</code>	try: <code>print(x)</code> except: <code>print(y)</code> else: <code>print("z")</code>	try: <code>print("X")</code> except: <code>print("y")</code> finally: <code>print("fish")</code>	try: <code>f = open("demo.txt")</code> <code>f.write("Lorum Ipsum")</code> except: <code>print("Something went wrong when writing to the file")</code> finally: <code>f.close()</code>
try ... except	using a named error	if there's no errors	finally run no matter what	example with file access

File Handling - Open: `open(filename, mode)`

"r"=read, "a"=append, "w"=write, "x" = create "t" text mode (default), "b" binary (for images)

<code>f = open("demo.txt")</code>	<code>f = open("demo.txt", "r")</code> <code>print(f.read())</code>	<code>print(f.read(5))</code>	<code>print(f.readline())</code>
<code>f = open("demo.txt", "rt")</code>			
read a file; default is "rt"	reading a file	read parts of the file - first 5 chars	read one line at a time
for x in f: <code>print(x)</code>	< read one line at a time entire file.		

File Handling - Write: `file.write("text")` Delete a File

"r"=read, "a"=append, "w"=write, "x" = create "t" text mode (default), "b" binary (for images)

<code>f = open("demo.txt", "x")</code>	<code>f = open("demo.txt", "w")</code>	<code>import os</code> <code>os.remove("demo.txt")</code>
creates a file; error if file exists already	creates/overwrites a file	remove a file.
<code>import os</code> <code>if os.path.exists("demo.txt")</code> <code> os.remove("demo.txt")</code> <code>else:</code> <code> print("No file")</code>	<code>import os</code> <code>os.rmdir("myfolder")</code>	