Python (v3.6)

Notes open for creative commons use @ developer blog: https://unfoldkyle.com, github: SmilingStallman, email: kmiskell@protonmail.com

Learning Resources
-Primary: Python Official Docs: https://docs.python.org/3/tutorial/index.html
-In conjunction with: *"Python Crash Course"* (2019)

Intro – Variables, Strings, & Lists

About Python
-Interpreted scripting language (no compiler needed) with OOP capabilities, excellent for automation, with large collection of micro-frameworks and extensions

-Syntax style is very minimalist (no brackets, indentations), high-level language, with variety of built in data types. Less development time than C/C++/Java.

-Modular, enables easy re-usability and easy import of existing collections, which can be used as program basis and help follow design patterns

-named after *Monty Python's Flying Circus*

Theory
-Beautiful code is better than ugly code
-Simple is better than complex
-Complex is better than complicated (many components better than high level of difficulty)
-Readabilty counts
-Do it the obvious way that would make most sense to programming standards, designs, etc.
-Now is better than tomorrow, which is often never. Always keep learning.

Setup
-python3 and python 2 come installed by default on most linux

-Will want to install pip package manager: *sudo apt-get install -y python3-pip*
-Install with: *pip3 install package_name*

-Extras: *sudo apt-get install build-essential libssl-dev libffi-dev python-dev*

Virtual Environments
-isolated python namespace from rest of OS, ensuring each project has own set of dependencies, etc., useful work working with diff versions, third party packages, etc.
-*pip* installed packages in 1 env are not installed in others

-Install v-environment packages: *sudo apt-get install -y python3-venv*

-Envs are directory based, with a couple scripts added to dir by *python3-venv* to set up env

-To create env, from folder want project in, run: *python3 -m venv my_env_name*
-Will generate bin, include, etc. dirs & files, similar to what *create-react-app* does with *node_modules*

-Enter env: *source my_env/bin/activate*
-Exit env: *deactivate*

Python Interpreter
-Enter interpreter terminal via *python3* command
-Allows interactive editing and execution of code real-time

-If program contains errors, run through interpreter on terminal to get errors printed

Basic Script
-folder/file naming convention: *my_script.py*

-Run from command line via: *python my_script.py*

Variables
-syntax: *name = value*
  -ex. *message = "hello world"*

-note no type, no *$* to signify var, etc.
-naming convention: underscore, letters, nums only. Lowercase. Cannot start with num. *my_var*

-Variables in Python can be thought as labels: labels you can assign to values. Stored by reference.

-in interactive terminal, last printed expression result is store in temp _ variable:

```
>>> 3 * 3
9
>>> 5 + _
14
```

-Reference to undefined variable will throw error

Comments
some_code   #comment

#multi-line
#comment

#do actually include meaningful comments, to help with quick comprehension for others reading code

Multiple Assignment
-Can assign multiple variables with multiple values in one statement with comma separation
-syntax: *x, y, z = 1, 2, 3*

-Must have value for each variable. Cannot just do *x, y, z = 0*

Constants
-No built in *const* variable but if want a variable to be a count give *ALL_CAPS_NAME* to indicate this to other devs (lol)

Numbers & Math Operators
-Typical math operands: *, / _, -, %,* etc.
-Group with *( )* for order of operations

-Division with / always returns a float, even if 1.0, etc.
-Drop decimal (floor) with // ex. *5 // 2 = 2*

-Powers: *2**3 // == 8*

-Any operation with mixed type operands (ex. *float* and *int)* will result in *float*

-Round to num decimal point with global funct: *round(round_me, decimal_points)*
  -ex. *round(126.18273, 2) //126.18*

-To make numbers more readable, can put underscores in difintions to represent commons. Does not change number value, but easier for human to read.
  -ex. *universe_age = 14_000_000_000 //universe_age === 14000000000*

Strings
-Single quote pairs or double quote pairs
-Escape non-paired quote with \

-*n* – new line escape char
-*t* – adds tab to string...useful for plaintext lists and text nesting

-If wrap strings in triple single or double quote pairs (‘’’ or “””), can span multiple lines and preserve indentations

-Concatenation: +
-Can concat with + to strings retured from expressions, stored by variables, etc.

-Strings next to each other auto concat:
  -ex. *my_string = (‘the’ ‘ string’*
  ‘is concated’) //the string is concated

-Can only auto-concat with string literals, not strings assigned to variables, returned from expressions, etc.

-Repeat string *x* times: *x * ‘string’*
  -ex. *3 * ‘string’ //stringstringstring*

String Indexing
-Can <u>read</u> (only) chars in string by referencing index as if array of chars
  my_string = ‘Hello’

my_string[0] //H

-Access from end index back via negative nums
my_string[-1] //o

-Index out of bounds attempt will throw error

String Slicing
-Can copy out substring as if pulling range from char array
-Starting num is inclusive, end is exclusive
my_string[2:4] //ll //chars 2, 3 as exclusive end

-If leave end off of slice (still using :), will count missing side as start or end automatically
my_string[:4] //llo

-Can also slice using negative index
my_string[-4:0] //

-To slice to end char using negative chars use default (*my_string[-4:])*, not *0*

-Index out of bounds will use default end or start instead for out of bounds end instead of error

-As Python variables are stored by reference, strings are immutable, and thus indexes cannot be changed via *my_string[2] = 'a'*, etc.
-Variables of course, can be set to reference other strings and variable name bindings are mutable

-String Length: *len(my_string)* // 5

String Casing Methods
my_string = ' this is my string '

-Uppercase words: *my_string.title( )* // *This Is My String*
-Uppercase whole string: *my_string.upper()* // *THIS IS MY STRING*
-lowercase whole string: *my_string.lower()* // *this is my string*

-*lower()* useful for normalization

f-strings
-The equivalent of template literals in JS, allowing variables to be reference by name, then output by value as part of the string. Aka. format strings

-syntax: *f"Hello my name is {name}" //f before quotes and reference in { }*

-Can use with any type of quote pair, including triple pairs
-*{ }* can hold variable name or expression/function output

-Python 3.6 addition. If using early version, need to use *format()* method:
"My name is {}. This {} tale".format("Ishmael", "is my") //args passed in order to {}

Removing Whitespace
-Removing all whitespace before records stored, compared, etc. useful for normalization

-Remove rightward whitespace: *my_string.rstrip()*
-Remove leftward whitespace: *my_string.lstrip()*
-Remove whitespace both sides: *my_string.strip()*

-As strings are immutable in python, these methods return new strings. To change ref of variable holding string: *mystring = my_strip.rstrip()*

Strings – See also
https://docs.python.org/3/library/stdtypes.html#textseq //text sequence strings
https://docs.python.org/3/library/stdtypes.html#string-methods //string methods
https://docs.python.org/3/reference/lexical_analysis.html#f-strings //formatted string literals
https://docs.python.org/3/library/string.html#formatstrings //format string syntax
https://docs.python.org/3/library/stdtypes.html#old-string-formatting //printf string formatting

Lists
-Access via index, as with array
my_list = [1, 4, 9, 'string', 1.92] *//Loosely typed and can hold mix of types*

-To view list contents simply *print(my_list)*

-List index access and slicing works the same as *string* slicing (see *String Slicing* section above). Lists are mutable, unlike strings though:
my_list[3] = 'fourth'

-List length - global function: *len(my_list)*

Can create nested listed
my_nested = [[1, 8, 12], ['a', 'b', 'c']]
my_nested[1][2] //c

List Range Assignment
-Can also set members using slice style range syntax with =
my_list[2:4] = [] *// = [] removes item*
print(my_list) *//[1, 'string', 1.92]* *//right is exclusive, as with string slice*

List Add/Remove Methods
-These methods, like setting range *[2:5]*, etc. directly modify the lists. No re-assignment is needed like with immutable strings. Calling *my_list.pop()* results in *my_list* having one less index.

-Append to end of list: *my_list.append(arg_x)*
-Delete item from list: *del my_list[2]*

-Remove end item: *my_list.pop()*
-*pop()* returns item popped, so can *pop()* and store in variable, etc.
-Can pass index num to remove to *pop()* any index: *my_list.pop(4)* *//index 4 popped*

-*del* more efficient than *pop()*, so if no need to access item on removal, use *del*

-Remove first occurrence of *arg*: *my_list.remove(arg)*

-Append index into list, pushing existing index forward 1: *my_list.insert(index, arg_x)*
-ex. *my_list = [0, 1, 2, 3, 4]*
my_list.insert(2, 'hello') *//[0, 1, 'hello', 2, 3, 4]*

Basic Control Flow Statements

No Brackets
-No brackets for blocks in python. Blocks built from tabs/space.
-All expressions in block must have same indentation:

```
for loop #1
  for loop #2
    do this            //same indentation
    also do this       //same indentation
print(something)
```

***if* Statements**
-*if, elif, else* with no ( ), and *:* at end of 'condition line'

```
if x < 0:
  x = 0
  print('x < 0')
elif x > 0:
  print('x > 0'
else:
  print('x == 0')
```

-Can use in same way as *switch* or *case* statements in other langs

***for* Statements**
-Iterate using *for index in iterable:*, where each loop *index* holds current index in *iterable* and loop ends when list ends

-Syntax:
```
for index in iterable:
    index stuff to do
```

-ex. *ints = [1, 2, 5, 9]*
```
for int in ints:
    print( int )    //1, 2,5,9
```