EN.540.635 "Software Carpentry"

Python Cheatsheet

Python has many definitions. Below I summarize a few of the common definitions Python goes by. Python is a :

1. **General purpose** : Can build software for a wide range of applications. Other examples are C, C++ and Java. On the contrary SQL is a domain specific language for querying databases. CSS and HTML are specific to creating websites.
2. **High Level** : Provides strong abstraction from machine level details.
3. **Object Oriented** : A type of programming paradigm where data and logic are store together in virtual containers called objects. Objects are created from classes which serve as the blueprint for objects. Everything in Python is built using objects and the program defines the manipulation between objects.
4. **Dynamic typed** : The type of objects are determined during runtime. Unlike in other languages, this allows the programmer to forgo declaring the type of variable at the start of the program.

To translate the high level code written by the programmer, Python uses both an interpreter and a compiler. An interpreter converts high level code into machine code and executes the program line by line while a compiler converts the high level code into machine code all at once. To access the Python interpreter shell, open a terminal/shell on your computer and type in python at the command line. Depending on the version of Python installed on your system you should see a window similar to the one below. Typing a command at the >>> and hitting enter executes the command and returns the output on the next line.

```
Python 3.8.11 (default, Jul 29 2021, 14:57:32)
[Clang 12.0.0 ] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print('Hello world!')
Hello world!
```

While the interpreter shell is useful for debugging small bits of code or testing out new Python functionality, it becomes tedious to write large codes and share them. Python programs are often written as scripts in text files using editors like vi, emacs, nano, sublime etc. and stored with the file extension '.py'. To try this open an editor and type in the above code and store it as 'my_first_python_script.py'. To execute the above program open the terminal and navigate to the directory the python script is stored at and type python 'my_first_python_script.py'.

```
user_name@machine_name ~ $python my_first_python_script.py
```

In the background what happens is the python source code is checked for any errors and if there are no errors it gets compiled into an intermediate representation called python bytecode (has extension .pyc) The Python bytecode is a machine agnostic representation. This means that the code can be run on any computer provided the appropriate Python Virtual Machine (PVM) is installed. This virtual machine is the Python Interpreter. Normally this file is hidden from view when you type python 'my_first_python_script.py'. To view this file type in the following command in the terminal.

```
user_name@machine_name ~ $python -m py_compile my_first_python_script.py
```

This should create a directory called __pycache__ in your current directory. Inside the directory there will be a file called my_first_python_script.cpython-38.pyc The contents of the file are all gibberish as they are just a series of bytes. Luckily we can make sense of the content of the file by using a python module called dis which 'disassembles' the python bytecode into human readable format. To do this run the following command in the terminal.

```
user_name@machine_name ~ $python -m dis my_first_python_script.py
```

The above command outputs the instructions fed to the Python interpreter in assembly code. The CPython interpreter operates using stack data structures. Refer to https://opensource.com/article/18/4/introduction-python-bytecode and dis documentation to read the details of what each instruction does.

Python built-in types

Everything in Python is an object. These objects come in several types. The principle built-in types are numerics, sequences, mappings, classes, instances and exceptions. type() can be used to check the type of data you are dealing with.

```
>>> a = 1
>>> type(a)
<class 'int'>
```

This cheatsheet summarizes the built-in types and functions supported by the Python interpreter. The material is sourced from The Python Standard Library.

1. Numeric Type

These represent all numerical data and variables created with these data.

Type	Description	Example usage
int	Represents any integer-valued number.	>>> a = 1
float	Represents any real-valued number.	>>> a = 1.0

complex	Represent any complex-valued numbers.	>>> a = complex(real=1, imag=1)
		>>> print(a)
		(1+1j)

Supported operations with numeric types	**Description**	**Example usage**
x + y	Addition	>>> 2 + 3 # Output : 5 >>> a += 5 # a=a+5 (Aug. Assign)
x - y	Subtraction	>>> 2 − 3 # −1 >>> a −= 5 # a = a − 5
x * y	Multiplication	>>> 2 ∗ 3 # 6 >>> a ∗= 5 # a = a ∗ 5
x / y	Division	>>> 2 / 3 # 0.666 >>> a /= 5 # a = a / 5
x // y	Floored Quotient	>>> 3 // 2 # 1 >>> a //= 5 # a = a // 5
x % y	Remainder of x/y	>>> 5 % 2 # 1 >>> a %= 5 # a = a % 5
x ** y (or) pow(x,y)	Power	>>> 3 ∗∗ 2 # 9 >>> a ∗∗= 5 # a = a ∗∗ 5
abs(x)	Absolute value	>>> abs(−3) # 3
int(x)	Type casting operation to int type. Similar to math.floor() or math.ceil().	>>> int(−3.56) # 3
float(x)	Type casting operation to float type.	>>> float(−3) # −3.0

3. String Type

These represent all the textual data in Python. Strings can be written using 'single quotes' or "double quotes" or """triple quotes""". Triple quoted strings can span multiple lines and includes the white space in between. They are most often used when writing doc strings.

String methods	Description	Example usage

str.capitalize()	First letter capitalized and rest are lowercased.	>>> a='hello'
		>>> a.capitalize()
		'Hello'

str.title()	Return a string with title case (first character uppercase and remaining characters lowercase)	>>> a='hello python!' 'Hello Python!'
str.count('substring')	Counts the number of occurrences of the substring	>>> a.count('l') 2

str.find('substring')	Returns the lowest index of the location of the substring. To check if a given substring is present in the string use the 'in' operator	>>> a.find('lo')
		3
		>>> 'lo' in 'hello'
		True

f-strings	Use to format a given string and replace text within a string denoted by {}	>>> print(f"{a} Python !") 'hello Python !'
str.join(iterable)	Joins an iterator (can be list, tuple, dict) of strings. The string on which you call will be the separator. In the example a blank space is a separator between the strings.	>>> ' '.join(['Hello ',' Python ','!']) 'Hello Python !'
str.split(sep)	Splits a string at the separator.	>>>a='Hello, Python!' >>>a.split(',') ['Hello', ' Python!']
str.strip([chars])	Useful for removing trailing and leading whitespaces if characters to strip are not specified. Can use str.lstrip([chars]) or str.rstrip([chars]) to remove only the leading and trailing characters respectively.	>>>a=' hello ' >>>a.strip() 'hello'
Supported Operations with string types	**Description**	**Example usage**
'str1' + 'str2'	String Concatenation	>>>'str1' + 'str2' 'str1str2'
constant*'str1'	Multiplication with a constant	>>>3*'str1' 'str1str1str1'

3. Boolean Type

These represent the truth values. These are the two constants True and False. They are used in conditional statements *if..elif..else* and in for and while loops.

Supported logical operations with boolean types	**Description**	**Example usage**
and	This is the equivalent of and gate. Truth table : True and True = True True and False = False False and True = False False and False = False	>>>a = 1 >>>b = 2 >>>if a.is_integer() and b. is_integer (): ... print('a and b are integers') 'a and b are integers'
or	This is the equivalent of or gate. Truth table : True or True = True True or False = True False or True = True False or False = False	>>>a = 1 >>>b = 2.0 >>>if a.is_integer() or b. is_integer (): ... print('Either a or b are integers') 'Either a or b are integers'
not	This is the equivalent of not gate. Truth table : not True = False not False = True	>>>a = 1.0 >>>if not a.is_integer (): ... print('a is not integer') 'a is not integer'
is	This is called the object identity operator. The example to the right shows an equivalent way of writing the above example using the combination of 'is' and 'not' operators.	>>>a = 1.0 >>>if type(a) is not int: ... print('a is not integer') >>>else: ... print('a is integer') 'a is not integer'

3. Sequence Type

There are three basic sequence types in Python : list, tuple, range and strings. Among sequences, lists and strings are called mutable sequences.

Type	Description	Example usage
list	An iterator object that can store any object. Can be created by : 1. Using pair of square brackets : [] 2. Using type constructor : list() 3. Using list comprehension : [x for x in iterable] Properties of a list : 1. Can modify contents of the list without creating a new list (mutability). 2. Can store objects of different types. Useful when you need to store related data under a single object.	>>>[1,2,3] [1,2,3] >>>list('123') ['1', '2', '3'] >>>[x for x in range(1,4)] [1, 2, 3] >>>cool_list[0]=4 >>>['Hello',4, cool_list] ['Hello', 4, [1,2,3]]
tuple	An iterator object that can store any object. Can be created by : 1. Using pair of parentheses : () 2. Using type constructor : tuple() Properties of tuple : 1. Cannot modify contents of tuple without creating a new tuple. (immutability) 2. Can store objects of different types. Useful when you dont want to modify the contents of original database. Tuples are created when using the built-in enumerate() function in Python.	>>>(1,) # Singleton tuple (1,) >>>(1,2,3) (1,2,3) >>>tuple([1,2,3]) #Creation from list (1,2,3) >>>for data in enumerate(cool_list): ... print(data[0], data[1])
range	Creates an immutable sequence of numbers useful for looping in 'for' loops	>>>list(range(1,4)) [1,2,3]
set	An unordered collection of distinct objects. Commonly used for membership testing, removing duplicates from a sequence, computing mathemical operations like intersection, union, difference, and symmetric difference	>>>duplicates=[1, 2, 3, 1, 2, 4, 6] list(set(duplicates)) [1, 2, 3, 4, 6]

Supported Operations with sequence types	Description	Example usage

seq1 + seq2	Sequence Concatenation. Best for lists and strings. Concatenating tuples results in a new tuple (due to immutability) (Exercise: Check this is true using id() function in Python) resulting in increased runtime. Not supported for range.	>>>cool_list = [1]
		>>> cool_list += [2]
		>>> cool_list
		[1,2]

constant*seq	Replicating the sequence	>>>3∗['hello'] ['hello', 'hello', 'hello']
x in seq / x not in seq	Truth testing to determine if desired object is present or not in seq.	>>>1 in [1, 2, 3'] True
seq[i:j] and seq[i:j:k]	Slicing operation	>>>seq=[1, 2, 3, 4, 5, 6] >>>seq[0:2] [1, 2, 3] >>>seq[0:−1:2] [1, 3, 5]
len(seq)	Length of sequence	>>>len(seq) 6
max(seq)	Largest item in sequence	>>>max(seq) 6
min(seq)	Smallest item in sequence	>>>min(seq) 1

Some methods that work with lists :

1. list.append(x) : Add an item to end of list.
2. list.extend(iterable) : Extend the current list with all objects from the other list.
3. list.insert(i, x) : Insert object 'x' at position 'i'.
4. list.remove(x) : Remove first item from list whose value is x.
5. list.sort() : Sort the list
6. list.reverse()

4. Mapping Type

The standard mapping type in Python are Dictionaries. They have a very different structure compared to sequences. Unlike in sequence types which are indexed by numbers, dictionaries are indexed by keys. Keys can be any immutable type. Can also use strings and numbers as keys. Only requirement is keys must be unique. For every key there is a value which can be any type of object. A useful analogy to think about dictionaries is like an excel sheet. The column headers are the keys and the rows for each column are the values. Lets look at how we can create a dictionary to store the heights of students in a class.

```
>>>student_heights={'name':['Jack', 'Jill', 'Bob', 'Tony', 'Alice']
                   'heights (cms)':[180, 170, 172, 168, 182]}
>>>student_heights = dict([['Jack', 'Jill', 'Bob', 'Tony', 'Alice'], [180, 170, 172, 168, 182]]) #
    Build from nested lists
>>>list(student_heights.keys()) # To get list of all keys
```

```
['name', 'heights (cms)']
>>>list(student_heights.values()) # To get a nested list
[['Jack', 'Jill', 'Bob', 'Tony', 'Alice'], [180, 170, 172, 168, 182]]
>>>len(student_heights) # Number of keys
2
>>>sum(list(student_heights.values())[1]) # Sums all the heights of the students
872
```