Python Basics Cheat Sheet

For More Details Visit at http://codewarriors2020.github.io/

CODE WARRIORS

Variables and Strings

Variables are used to store values. A string is a series of characters, surrounded by single or double quotes. Strings are immutable.

Hello World

```
print(“Hello World”)
```

Hello World with variables

```
variable_name = “Code Warriors”
print(“variable_name”)
```

Strings Concatenation

```
msg_1 = “Learn”
msg_2 = “Coding with Code Warriors”
full_msg = msg_1 + “ ” + msg_2
print(full_msg)
```

Lists

List can be written as a list of comma-separated values (items) between square brackets.

Creating a list

```
list = ['physics', 'chemistry', 1997, 2000]
```

Accessing Values in Lists

```
print("list[0]: ", list[0])          #list[0]: physics
print("list[1:3]: ", list[1:3])      # list[1:4]:
chemistry,1999
```

Deleting List Elements

```
del list[2]
```

Basic List Operations

Python Expression	Results	Description
len([1, 2, 3])	3	Length
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	Concatenation
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	Repetition
3 in [1, 2, 3]	True	Membership
for x in [1, 2, 3]: print x,	1 2 3	Iteration

Lists (conti.)

List Comprehensions

```
squares = [x**2 for x in range(10)]
```

Indexing, Slicing, and Matrixes

```
L = ['spam', 'Spam', 'SPAM!']
```

Python Expression	Results	Description
L[2]	'SPAM!'	Offsets start at zero
L[-2]	'Spam'	Negative: count from the right
L[1:]	['Spam', 'SPAM!']	Slicing fetches sections

Tuple

A tuple is an immutable sequence of mutable/ immutable python objects. Tuples are sequences, just like lists.

Creating Tuple

```
tup1 = ('physics', 'chemistry', 1997, 2000)
tup2 = (1, 2, 3, 4, 5 )
tup3 = 'a', 'b', 'c', 'd'
tup4 = (1,"a",[3,4],(1,2,3))
print(tup4[2])        # [3,4]
print(tup4[2][0])     # 3
```

Dictionaries

Dictionary is one of the data type in python. It is mapping between key and values. The association of a key and a value is called a key value pair or sometimes an item.

Creating Dictionary

```
a = dict()
a = {}
```

Adding & Accessing items

```
a[1] = ‘abc’
a[‘a’] = ’abc’
a['b'] = 'pqr'
print(a[‘a’])               #pqr
```

Looping in dictionaries

```
counts = { 1:'abc',2:'pqr',3:'xyz'}
for key in counts:
        print (key, counts[key])
```

Dictionary (conti.)

Updating dictionary

```
a = {'a': 'abc', 'b': 'pqr'}
a[‘a’] = ’zxc’
```

If Statements

If statements are used to test for particular conditions and respond appropriately.

Conditional tests

equal	x == 16
not equal	x != 16
greater than	x > 16
or equal to	x >= 16
less than	x < 16
or equal to	x <= 16

Conditional test with lists

```
‘maths’ in list
‘maths’ not in list
```

A Simple if test

```
if a>=16:
        print(‘greater than 16’)
```

if-elif-else statement

```
if a>16:
        print(‘a is greater than 16’)
elif a<16:
        print(‘a is smaller than 16’)
else:
        print(‘a is equal to 16’)
```

Input

Program can prompt the user for input. All input is stored as a string. We can type cast it.

Prompting for a value

```
name = input(“Enter your name”)
print(“Your name is ”,name)
```

Prompting for integer value

```
value = int(input(“Enter any integer value”))
print(“Value entered by you ”,value)
```

Prompting for float value

```
value = float(input(“Enter any integer value”))
print(“Value entered by you ”,value)
```

Functions

A block of organized, reusable code that is used to perform a single, related action.
Provides better modularity for your application and a high degree of code reusing.
Python gives you many built-in functions like print(), etc. but you can also create your own functions. These functions are called user-defined functions

Defining a Function

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

- Function blocks begin with the keyword def followed by the function name and parentheses (( )).
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or docstring.
- The code block within every function starts with a colon (:) and is indented.
- The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None

Making a function

```
def func():
        """Creating a simple function"""
        print("Welcome to Code Warriors!!")
func()
```

Pass by Reference vs Value

- All parameters (arguments) in the Python language are passed by reference.
- Means if you change what a parameter refers to within a function, the change also reflects back in the calling function.

```
def changeme( mylist ):
        "This changes a passed list into this
        function"
        print ("Values inside the function before
        change: ", mylist)
        mylist[2]=50
        print ("Values inside the function after
        change: ", mylist)
        return
mylist = [10,20,30]
changeme( mylist )
print ("Values outside the function: ", mylist)
```

Function Arguments

You can call a function by using the following types of formal arguments −

- Required arguments
- Keyword arguments
- Default arguments
- Variable-length arguments

Required Arguments/ Positional Arguments

- Required arguments are the arguments passed to a function in correct positional order.
- Here, the number of arguments in the function call should match exactly with the function definition.

```
def printme( str ):
        "This prints a passed string into this
        function"
        print (str)
        return
printme()
```

Keyword Arguments

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

```
def printme( str ):
        "Prints a passed string into this function"
        print (str)
        return
printme( str = "My string")
```

Default Arguments

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument.

```
def printinfo( name, age = 35 ):
        "Prints a passed info into this function"
        print ("Name: ", name)
        print ("Age ", age)
        return
printinfo( age = 50, name = "miki" )
printinfo( name = "miki")
```

Variable-Length Arguements

- You may need to process a function for more arguments than you specified while defining the function.
- These arguments are called variable-length arguments and are not named in the function definition, unlike required and default arguments.

```
def printinfo( arg1, *vartuple ):
        "This prints a variable passed arguments"
        print ("Output is: ")
        print (arg1)
        for var in vartuple:
                print (var)
        return
printinfo( 10 )
printinfo( 70, 60, 50 )
```

Anonymous Functions

```
lambda
# Function definition is here
sum = lambda arg1, arg2: arg1 + arg2
# Now you can call sum as a function
print ("Value of total : ", sum( 10, 20 ))
```

Files and Exception

A file is some information or data which stays in the computer storage device. Python gives you easy way to manipulate these files. Generally we divide files in two categories, text file and binary file. Text files are simple text whereas the binary files contain binary data which is only readable by computer.
An exception is an error that happens during execution of a program. When that error occurs, Python generate an exception that can be handled, which avoids your program to crash.

File Opening

To open a file we use *open()* function. It requires two arguments, first the file path or file name, second which mode it should open. Modes are like

- "r" -> open read only, you can read the file but can not edit / delete anything inside
- "w" -> open with write power, means if the file exists then delete all content and open it to write
- "a" -> open in append mode

The default mode is read only, ie if you do not provide any mode it will open the file as read only. Let us open a file

```
fobj = open("code.text")
fobj
<_io.TextIOWrapper name ='code.txt' mode='r'
encoding='UTF-8'>
```

Closing a File

After opening a file one should always close the opened file. We use method close() for this.

```
fobj = open("code.text")
fobj
<_io.TextIOWrapper name ='code.txt' mode='r'
encoding='UTF-8'>
fobj.close()
```

Note : Always make sure you *explicitly* close each open file, once its job is done and you have no reason to keep it open.

Reading a File

To read the whole file at once use the read() method.

```
fobj = open('sample.txt')
fobj.read()
'I love Python\n Snehal loves Android\n'
```

If you call read() again it will return empty string as it already read the whole file. readline() can help you to read one line each time from the file.

```
fobj = open('sample.txt')
fobj.readline()
'I love Python\n
fobj.readline()
Snehal loves Android\n'
```

To read all the lines in a list we use readlines() method.

```
fobj = open('sample.txt')
fobj.readlines()
['I love Python\n Snehal loves Android\n']
```

Using the with statement

In real life scenarios we should try to use with statement. It will take care of closing file for you.

```
with open('setup.py') as fobj:
        for line in fobj:
                print(line)
#!/usr/bin/env python3
"""Factorial project"""
from setuptools import find_packages, setup
setup(name = 'factorial',
    version = '0.1',
    description = "Factorial module.",
    long_description = "A test module for our
book.",
    platforms = ["Linux"],
    author="Kushal Das",
    author_email="kushaldas@gmail.com",
url="https://pymbook.readthedocs.io/en/latest/",
    license =
"http://www.gnu.org/copyleft/gpl.html",
    packages=find_packages()
    )
```

Writing in a File

Let us open a file then we will write some random text into it by using the *write()* method. We can also pass the file object to the print function call, so that it writes in the file.

```
fobj = open("ircnicks.txt", 'w')
fobj.write('coding with\n')
fobj.write('code\n')
fobj.write('warriors\n')
fobj.write('team')
print("This is the last line.", file=fobj)
fobj.close()
```

Try-Except Block

The words "try" and "except" are Python keywords and are used to catch exceptions.
• The code within the try clause will be executed statement by statement.
• If an exception occurs, the rest of the try block will be skipped and the except clause will be executed.
• try:
  some statements here
• except:
  exception handling

```
try:
        print (1/0)
except ZeroDivisionError:
        print ("You can't divide by zero, you're
        silly.")
```

Try ... except ... else clause

```
try:
        fh= open("testfile","w")
        fh.write("This is my test file for exception
handling!!")
except IOError:
        print("Error: can\'t find file or read data")
else:
        print("Written content in the file
successfully")
fh.close()
```

OBJECT-ORIENTED

- Python is an object-oriented language.
- Due to that creation and usage of class and object is very much easy.
- You can implement any of the OOP concepts using python.

Overview of OOP Terminology

- **Class**: A user-defined prototype for an object that defines a set of attributes that characterise any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.
- **Instance**: An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle.
- **Class variable**: A variable that is shared by all instances of a class.
- **Instance variable**: A variable that is defined inside a method and belongs only to the current instance of a class.
- **Data member**: A class variable or instance variable that holds data associated with a class and its objects.
- **Instantiation**: The creation of an instance of a class.
- **Function overloading**: The assignment of more than one behaviour to a particular function. The operation performed varies by the types of objects or arguments involved.
- **Operator overloading**: The assignment of more than one function to a particular operator.

Creating Classes

The class statement creates a new class definition. The name of the class immediately follows the keyword class followed by a colon as follows:

```
class ClassName:
        'Optional class documentation string'
        class_suite
```

The class has a documentation string, which can be accessed via ClassName.__doc__.

```
class Employee:
        'Common base class for all employees'
        empCount = 0
        def __init__(self, name, salary):
                self.name = name
                self.salary = salary
                Employee.empCount += 1
        def displayCount(self):
                print ("Total Employee %d" %
                Employee.empCount)
        def displayEmployee(self):
                print ("Name : ", self.name,  ",
                Salary: ", self.salary)
```

Creating object of a class

You can create an object of a class using any variable as an object followed by assignment of class name.
e.g.
myobjectx = MyClass()
emp1= Employee(“abc”,5000)
The variable "myobjectx" holds an object of the class "MyClass" that contains the variable and the function defined within the class called "MyClass".

Class Inheritance

You can create a class by deriving it from a preexisting class by listing the parent class in parentheses after the new class name.
The child class inherits the attributes of its parent class.
A child class can override data members and methods from the parent.

```
class Parent:
        # define parent class
        parentAttr = 100
        def __init__(self):
                print "Calling parent
        constructor"
        def parentMethod(self):
                print 'Calling parent method'
        def setAttr(self, attr):
                Parent.parentAttr = attr
        def getAttr(self):
                print "Parent attribute :",
                Parent.parentAttr
class Child(Parent):
        # define child class
        def __init__(self):
                print "Calling child constructor"
        def childMethod(self):
                print 'Calling child method'
```

Overriding methods

You can always override your parent class methods.

```
class Parent:
         # define parent class
        def myMethod(self):
                print 'Calling parent method'
class Child(Parent):
        # define child class
        def myMethod(self):
                print 'Calling child method'
c = Child()         # instance of child
c.myMethod()        # child calls ________method
```

Overloading Operators

Suppose you have created a Vector class to represent two-dimensional vectors, what happens when you use the plus operator to add them?

```
class Vector:
        def __init__(self, a, b):
                self.a = a
                self.b = b
        def __str__(self):
                return 'Vector (%d, %d)' %
                (self.a, self.b)
        def __add__(self, other):
                return Vector(self.a + other.a,
                self.b + other.b)
v1 = Vector(2,10)
v2 = Vector(5,-2)
```

Python Basics
Learn From Experts

CODE WARRIORS