FRANK A. - FRANK@PYTHONSPOT.COM

# AN INTRODUCTION TO PYTHON

PYTHONSPOT.COM

PUBLISHED BY PYTHONSPOT.COM

PYTHONSPOT.COM

*First printing, July 2016*

# Contents

Numbers

Text

Lists

Dictionary

Tuples

If statements

Loops

Functions

Classes

# *List of Figures*

# *Introduction*

Welcome to my Python Course! This course is suitable for both version 2 and 3.

Python is a general-purpose computer programming language, ranked among the top eight most popular programming languages in the world.

To run Python programs, you need the Python interpreter that you can download from https://python.org. Typically you would copy the code and save it as a text file with the extension .py. Execute using:

```
python filename.py
```

I recommend you to use an IDE, which makes programming a lot easier. You can find a list of IDEs here: https://pythonspot.com/en/python-ides/

# Part I

# Data types

# *Numbers*

## *Datatypes*

Variables in Python can hold numbers which are Integer, Float or Boolean. Integers are whole numbers (1,2,3,4), floats have numbers behind comma and a boolean is either True or False.

```
x = 1
y = 1.234
z = True
```

Output them to the screen using the print() function.

```
x = 1
y = 1.234
z = True

print(x)
print(y)
print(z)
```

Python supports arithmetic operations like addition (+), multiplication (*), division (/) and subtractions (-).

```
x = 3
y = 8

sum = x + y

print(sum)
```

## User Input

Use the input() function to get text input, convert to a number using int() or float().

```

x = int(input("Enter␣x:"))
y = int(input("Enter␣y:"))

sum = x + y
print(sum)
```

If you use Python 2.x, use this instead:

```
x = int(raw_input("Enter␣x:"))
y = int(raw_input("Enter␣y:"))

sum = x + y
print(sum)
```

## Videos

This book has a video series: Watch on Youtube

# *Text*

## *Output and Input*

Text in Python are known as strings. A string is essentially a collection of characters.

To output text to the screen:

```
s = "hello world"
print(s)
```

To get text from keyboard:

```
name = input("Enter name: ")
print(name)
```

## *Comparison*

To test if two strings are equal use the equality operator (==).

```
sentence = "The cat is brown"
q = "cat"

if q == sentence:
    print('strings are equal')
else:
    print('strings are not equal')
```

### *Slices*

Python indexes the characters of a string, every index is associated with a unique character. For instance, the characters in the string "python" have indices:

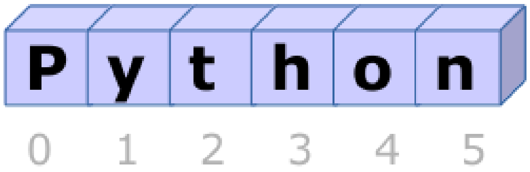

Figure 1: A string and its indices. Zero is the first index.

The 0th index is used for the first character of a string. Try the following:

```
s = "Hello␣Python"
print(s)       # prints whole string
print(s[0])    # prints "H"
print(s[1]) # prints "e"
```

Given a string s, the syntax for a slice is:

```
s[ startIndex : pastIndex ]
```

The startIndex is the start index of the string. pastIndex is one past the end of the slice. If you omit the first index, the slice will start from the beginning. If you omit the last index, the slice will go to the end of the string. For instance:

```
s = "Hello␣Python"
print(s[0:2]) # prints "He"
print(s[2:4]) # prints "ll"
print(s[6:])  # prints "Python"
```

### *Videos*

This book has a video series: Watch on Youtube

# Lists

Python has a datatype for lists, known as "list". A list may contain strings (text) and numbers. Sometimes lists are called arrays.

## Definition

Lists are defined using the brackets []. To access the data, these same brackets are used. Like strings, the first element is [0]. Example list usage:

```
l = [ "Drake", "Derp", "Derek", "Dominique" ]

print(l)    # prints all elements
print(l[0]) # print first element
print(l[1]) # prints second element
```

## Appending and removing

You can append and remove to a list with:

```
l = [ "Drake", "Derp", "Derek", "Dominique" ]

print(l)                # prints all elements
l.append("Victoria")    # add element.
print(l)                # print all elements
l.remove("Derp")        # remove element.
l.remove("Drake")       # remove element.
print(l)                # print all elements.
```

## *Sorting*

The sort() method can be used to sort a list:

```
l = [ "Drake", "Derp", "Derek", "Dominique" ]

print(l)     # prints all elements
l.sort()     # sorts the list in alphabetical order
print(l)     # prints all elements
```

To sort in decending order:

```
l = [ "Drake", "Derp", "Derek", "Dominique" ]

print(l)     # prints all elements
l.sort()     # sorts the list in alphabetical order
l.reverse()  # reverse order.
print(l)     # prints all elements
```

## *Videos*

This book has a video series: Watch on Youtube

# *Dictionary*

## *Definition*

A dictionary can be thought of as an unordered set of key: value pairs. A pair of braces creates an empty dictionary: . Each element can maps to a certain value. An integer or string can be used for the index. Dictonaries do not have an order. Let us make a simple dictionary:

```
words = {}
words["Hello"] = "Bonjour"
words["Yes"] = "Oui"
words["No"] = "Non"
words["Bye"] = "Au␣Revoir"

print words["Hello"]
print words["No"]
```

We are by no means limited to single word defintions in the value part. A demonstration:

```
dict = {}
dict['Ford'] = "Car"
dict['Python'] = "The␣Python␣Programming␣Language"
dict[2] = "This␣sentence␣is␣stored␣here."

print dict['Ford']
print dict['Python']
print dict[2]
```

### *Manipulating a dictionary*

We can manipulate the data stored in a dictionairy after declaration. This is shown in the example below:

```
words = {}
words["Hello"] = "Bonjour"
words["Yes"] = "Oui"
words["No"] = "Non"
words["Bye"] = "Au␣Revoir"

print words              # print key-pairs.
del words["Yes"]         # delete a key-pair.
print words              # print key-pairs.
words["Yes"] = "Oui!"    # add new key-pair.
print words              # print key-pairs.
```

# Tuples

## Definition

The tuple data structure is used to store a group of data. The elements in this group are separated by a comma. Once created, the values of a tuple cannot change. An empty tuple in Python would be defined as:

```
tuple = ()
```

A comma is required for a tuple with one item:

```
tuple = (3,)
```

The comma for one item may be counter intuitive, but without the comma for a single item, you cannot access the element. For multiple items, you do not have to put a comma at the end. This set is an example:

```
personInfo = ("Diana", 32, "New␣York")
```

The data inside a tuple can be of one or more data types such as text and numbers.

## Data access

To access the data we can simply use an index. As usual, an index is a number between brackets:

```
personInfo = ("Diana", 32, "New␣York")
print(personInfo[0])
print(personInfo[1])
```

If you want to assign multiple variables at once, you can use tuples:

```
name,age,country,career = ('Diana',32,'Canada','CompSci')
print(country)
```

On the right side the tuple is written. Left of the operator equality operator are the corresponding output variables.

## *Append to a tuple*

If you have an existing tuple, you can append to it with the + operator. You can only append a tuple to an existing tuple.

```
x = (3,4,5,6)
x = x + (1,2,3)
print(x)
```

## *Convert a tuple*

Tuple to list To convert a list to a tuple you can use the tuple() function.

```
listNumbers = [6,3,7,4]
x = tuple(listNumbers)
print(x)
```

You can convert an existing tuple to a list using the list() function:

```
x = (4,5)
listNumbers = list(x)
print(listNumbers)
```

Tuple to string If your tuple contains only strings (text) you can use:

```
person = ('Diana','Canada','CompSci')
s = '␣'.join(person)
print(s)
```

## *Sorting*

Tuples are arrays you cannot modify and donâĂŹt have any sort function. You can however use the sorted() function which returns a list. This list can be converted to a new tuple. Keep in mind a tuple cannot be modified, we simple create a new tuple that happens to be sorted.

```
person = ('Alison','Victoria','Brenda','Rachel','Trevor')
person = tuple(sorted(person))
print(person)
```

## *Videos*

This book has a video series: Watch on Youtube.

# Part II

# Control Flow

# *If statements*

## *Definition*

In Python you can define conditional statements, known as if-statements. Consider this application:

```
x = 3
if x < 10:
    print("x␣smaller␣than␣10")
else:
    print("x␣is␣bigger␣than␣10␣or␣equal")
```

If you set x to be larger than 10, it will execute the second code block. We use indentation (4 spaces) to define the blocks.
Symbols you can use are equality (==), greater than (>), smaller than (<) and not equal (!=).

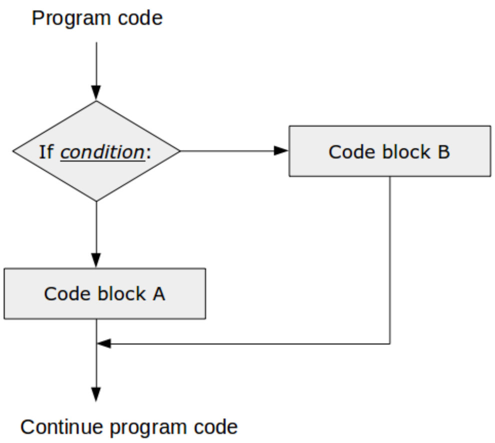

Figure 2: If statement, a visual Code Flow

### *A practical example*

A variable may not always be defined by the user, consider this little game:

```
age = 24

print "Guess␣my␣age,␣you␣have␣1␣chances!"
guess = int(raw_input("Guess:␣"))

if guess != age:
    print("Wrong!")
else:
    print("Correct")
```

### *Nesting*

The most straightforward way to do multiple conditions is nesting:

```
a = 12
b = 33

if a > 10:
    if b > 20:
        print("Good")
```

This can quickly become difficult to read, consider combining 4 or 6 conditions. Luckily Python has a solution for this, we can combine conditions using the and keyword.

```
guess = 24
if guess > 10 and guess < 20:
    print("In␣range")
else:
    print("Out␣of␣range")
```

Sometimes you may want to use the or operator.

# *Loops*

## *Definition*

In Python and many other programming languages you can repeat a part of code using a loop. A loop repeats a set of instructions N times. Python has 3 loops: for, while and nested loops.

We can iterate a list using a for loop

## *For loop*

A for loop may be used to repeat a part of the code, such as printing the elements of a list.

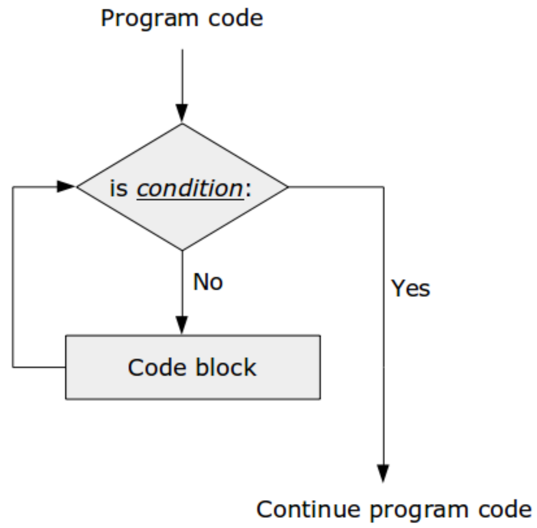

Figure 3: For loop, a visual Code Flow

```
items = [ "Abby","Brenda","Cindy","Diddy" ]

for item in items:
    print(item)
```

We can count using a for loop too:

```
for i in range(1,10):
    print i
```

### *Nested loop*

We can combine for loops using nesting. If we want to iterate over an (x,y) field we could use:

```
for x in range(1,10):
    for y in range(1,10):
        print("(" + str(x) + "," + str(y) + ")")
```

### *While loop*

Until a condition is met we can repeat some instructions. For example,

```
while button_not_pressed:
     drive()
```

We use this if we don't know exactly how many times the loop should be repeated, but we know the condition that the loop should stop:

```
i = 10

while i > 3:
    i = i - 1
    print(i)
```

# Functions

## Definition

A function is reusable code that can be called anywhere in your program. We use this syntax to define as function:

```
def function(parameters):
    instructions
    return value
```

The def keyword tells Python we have a piece of reusable code (A function). A program can have many functions.

## Calling and parameters

We can call the function using function(parameters).

```

def f(x):
    return x*x

print f(3)
```

The function has one parameter, x. The return value is the value the function returns. Not all functions have to return something. We can pass multiple variables:

```

def f(x,y):
    print 'You␣called␣f(x,y)␣with␣the␣value␣x␣=␣' + str(x) + '␣and␣y␣=␣' + str(y)
    print 'x␣*␣y␣=␣' + str(x*y)

f(3,2)
```

## Scope

Variables can only reach the area in which they are defined, which is called scope. This will not work:

```
def f(x,y):
    print('You␣called␣f(x,y)␣with␣the␣value␣x␣=␣' + str(x) + '␣and␣y␣=␣' + str(y))
    print('x␣*␣y␣=␣' + str(x*y))
    z = 4 # cannot reach z, so THIS WON'T WORK

z = 3
f(3,2)
```

but this will:

```
def f(x,y):
    z = 3
    print('You␣called␣f(x,y)␣with␣the␣value␣x␣=␣' + str(x) + '␣and␣y␣=␣' + str(y))
    print('x␣*␣y␣=␣' + str(x*y))
    print(z) # can reach because variable z is defined in the function

f(3,2)
```

Lets examine this further:

```
def f(x,y,z):
    return x+y+z # this will return the sum because all variables are passed as parameters

sum = f(3,2,1)
print(sum)
```

## *Calling functions from functions*

We can also get the contents of a variable from another function:

```
def highFive():
    return 5

def f(x,y):
    z = highFive() # we get the variable contents from highFive()
    return x+y+z # returns x+y+z. z is reachable becaue it is defined above

result = f(3,2)
print result
```

Another example:

```
def doA():
    a = 5

def doB(a):
    print a # we pass variable as parameter, this will work

doB(3)
```

In the last example we have two different variables named a, because the scope of the variable a is only within the function. The variable is not known outside the scope.

## *Global and local variables*

If a variable can be reached anywhere in the code is called a global variable. If a variable is known only inside the scope, we call it a local variable.

# Part III

# Object-Oriented Programming

# *Classes*

## *Introduction*

Technology always evolves. What are classes and where do they come from?

1. Statements: In the very early days of computing, programmers wrote only commands.

2. Functions: Reusable group of statements, helped to structure that code and it improved readability.

3. Classes: Classes are used to create objects which have functions and variables. Strings are examples of objects: A string book has the functions book.replace() and book.lowercase(). This style is often called object oriented programming.

Lets take a dive!

## *Objects and Classes*

We can create virtual objects in Python. A virtual object can contain variables and methods. A program may have many different types and are created from a class. Example:

```
class User:
    name = ""

    def __init__(self, name):
        self.name = name

    def sayHello(self):
        print "Hello,␣my␣name␣is␣" + self.name

# create virtual objects
james = User("James")
david = User("David")
eric = User("Eric")

# call methods owned by virtual objects
james.sayHello()
david.sayHello()
```

Run this program. In this code we have 3 virtual objects: james, david and eric. Each object is instance of the User class.

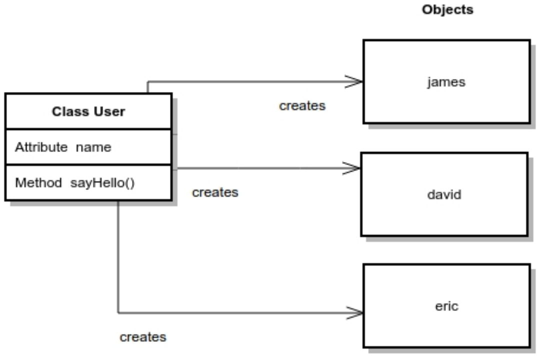

Figure 4: Objects created from a class

In this class we defined the sayHello method, which is why we can call it for each of the objects. The init() method is called the constructor and is always called when creating an object. The variables owned by the class is in this case "name". These variables are sometimes called class attributes.

## *Class variables*

We define a class CoffeeMachine of which the virtual objects hold the amount of beans and amount of water. Both are defined as a number (integer). We may then define methods that add or remove beans.

```
    def addBean(self):
        self.bean = self.bean + 1

    def removeBean(self):
        self.bean = self.bean - 1
```

We do the same for the variable water. As shown below:

```
class CoffeeMachine:
    name = ""
    beans = 0
    water = 0

    def __init__(self, name, beans, water):
        self.name = name
        self.beans = beans
        self.water = water

    def addBean(self):
        self.beans = self.beans + 1

    def removeBean(self):
        self.beans = self.beans - 1

    def addWater(self):
        self.water = self.water + 1

    def removeWater(self):
        self.water = self.water - 1

    def printState(self):
        print "Name␣␣=␣" + self.name
        print "Beans␣=␣" + str(self.beans)
        print "Water␣=␣" + str(self.water)

pythonBean = CoffeeMachine("Python␣Bean", 83, 20)
pythonBean.printState()
print ""
pythonBean.addBean()
pythonBean.printState()
```

Run this program. The top of the code defines the class as we described. The code below is where we create virtual objects. In this example we have exactly one object called âĂIJpythonBeanâĂİ. We then call methods which change the internal variables, this is possible because we defined those methods inside the class.

# *Method overloading*

## *Introduction*

In Python you can define a method in such a way that there are multiple ways to call it. This is known as method overloading. We do that by setting default values of variables.

## *Example*

Let us do an example:

```
class Human:

    def sayHello(self, name=None):

        if name is not None:
            print 'Hello␣' + name
        else:
            print 'Hello␣'

# Create instance
obj = Human()

# Call the method
obj.sayHello()

# Call the method with a parameter
obj.sayHello('Guido')
```

To clarify method overloading, we can now call the method sayHello() in two ways:

```
obj.sayHello()
obj.sayHello('Guido')
```

We created a method that can be called with fewer arguments than it is defined to allow. We are not limited to two variables, your method could have more variables which are optional.

# *Inheritance*

## *Introduction*

Classes can inherit functionality and variables from other classes. Visually that would look like:

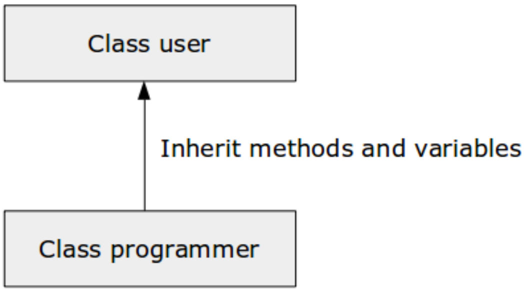

Figure 5: Inheritance

The variables and methods from the class user can be used in an object created from the class programmer.

Lets take a look at how that works. We start with a basic class:

```
class User:
    name = ""

    def __init__(self, name):
        self.name = name

    def printName(self):
        print "Name␣␣=␣" + self.name

brian = User("brian")
brian.printName()
```

This creates one instance called brian which outputs its given name. Add another class called Programmer.

```
class Programmer(User):

    def __init__(self, name):
        self.name = name
    def doPython(self):
        print "Programming␣Python"
```

This looks very much like a standard class except than User is given in the parameters. This means all functionality of the class User is accesible in the Programmer class.

### *Example*

Full example of Python inheritance:

```
class User:
    name = ""

    def __init__(self, name):
        self.name = name

    def printName(self):
        print "Name␣␣=␣" + self.name

class Programmer(User):
    def __init__(self, name):
        self.name = name

    def doPython(self):
        print "Programming␣Python"

brian = User("brian")
brian.printName()

diana = Programmer("Diana")
diana.printName()
diana.doPython()
```

Brian is an instance of User and can only access the method printName. Diana is an instance of Programmer, a class with inheritance from User, and can access both the methods in Programmer and User.