Reason for using Python

Syntax – easy to read

Community – Stackoverflow for answers

Indentation REQUIRED or syntax heavy (Line breaks as well)

Every languages has

Variables

Data types

Loops

Conditionals

And Functions

You place ‘’ or “” around a string as well as parentheses

Variables a is more like a name tag and it can refer to any value you want so when you say a = 1 it means a is referring to the value of 1

F = A would mean that f is referring to the value that A is referring to, not following it like a path but instead directly referring to the value A’s referring to

Using a single variable to switch the value of multiple variables

V1 = “first string”

V2 = “second string”

V1 = “first string”

V2 = “second string”

This would make both v1 and v2 refer to second string

V1 = “first string”

V2 = “second string”

Temp1 = v1

Temp2 = v2

Temp = v1

V1 = V2

V2 = temp

Type **int**(x) to convert x to a plain **integer**. Type long(x) to convert x to a long **integer**. Type float(x) to convert x to a floating-point number. Type complex(x) to convert x to a complex number with real part x and imaginary part zero.

A list is a data structure in **Python** that is a mutable, or changeable, ordered sequence of elements. Each **element** or value that is inside of a list is called an item.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| == | Checks if the value of two operands are equal or not, if yes then condition becomes true. | (1 == 2) is not true.  (1 == 1) is true. |
| != | Checks if the value of two operands are equal or not, if values are not equal then condition becomes true. | (1 != 2) is true. |
| <> | Checks if the value of two operands are equal or not, if values are not equal then condition becomes true. | (1 <> 2) is true. This is similar to != operator.\* |
| > | Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. | (1 > 2) is not true. |
| < | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. | (1 < 2) is true. |
| >= | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. | (1 >= 2) is not true. |
| <= | Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. | (1 <= 2) is true. |
| and | Checks each expression on the left and right. If both are true then this evaluates true. If either or both expressions are false then this is false | (1 <= 2 and 2 <= 3) is true.  (1 <= 2 and 2 >= 3) is false.  (1 >= 2 and 2 >= 3) is false. |
| or | Checks each expression on the left and right. If either of the expressions are true then this evaluates true. If both expressions are false then this is false. | (1 <= 2 or 2 >= 3) is true.  (1 <= 2 or 2 <= 3) is true.  (1 >= 2 or 2 >= 3) is false. |
| not | Reverses the true-false value of the operand | not(true) is false.  not(false) is true.  not(1 >= 2) is true.  not(1 =< 2) is false.  not(1 <= 2 and 2 =< 3) is false.  not(1 >= 2 or 2 >= 3) is true. |

A for loop is used to repeat a set of code over and over again and example is

For count in range (0, 5)

Print “looping – “, count

Desktop $ python sample.py (Output in terminal)

Looping – 0

Looping – 1

Looping – 2

Looping – 3

Looping – 4

# bool

int

str

float

#pass by value

myValue = 5

mySecondValue = myValue

myValue = 10

myNonWholeNumber = 10.5

myTrue = False

x = "abcd"

dict

list

tuple

#pass by reference

tree = {

"color\_of\_leaves":"green",

"age": 10

}

print tree["color\_of\_leaves"]

\_of\_tree = tree

tree["age"] = 15

print copy\_of\_tree

# 0 1 2 3

# myList = ("Minh", "AJ", "Chris", "Ghezal")

# myList[1] = "Anthony"

myList = ["Minh", "AJ", "Chris", "Ghezal"]

(myList[0], myList[1]) = (myList[1], myList[0])

x = 5

print len(myList)

*def* instruction1(name1, name2="Minh", name3):

print name1

print name2

print name3

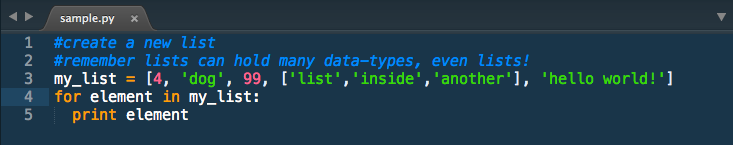
instruction1("Reena", , "Ghezal")

Python's *for* statement iterates over the items of any sequence(list or string), in the order they appear in the sequence. In the above example, we iterated through the range from 0 to 5 (exclusive) and printed out a 'looping - ' item in the sequence. Notice how we use *count* as a counter/variable to refer to the current item in our loop.

More generally, here's the basic syntax of a *for* loop:

for <counter> in <sequence or range>:

# do something



Here's a quick example of how you do that. If we execute this program, you'll see each value in our list printed.

4

dog

99

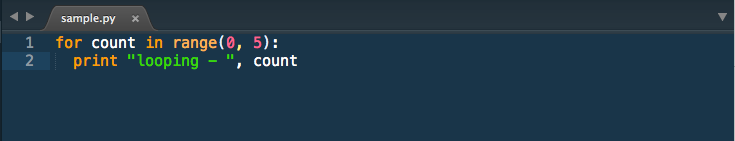
['list', 'inside', 'another']

hello world!

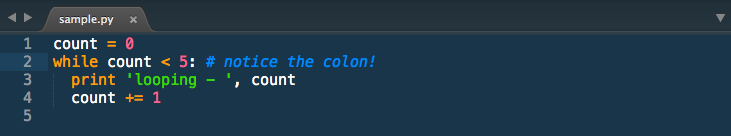
**While Loops**

**While** loops are often used when we *don't know how many times we have to repeat a block of code but we know we have to do it until a certain condition is met.*

Remember this *for* loop?



We can rewrite it as a *while* loop:



The basic syntax for a *while* loop looks like this:

while <expression>:

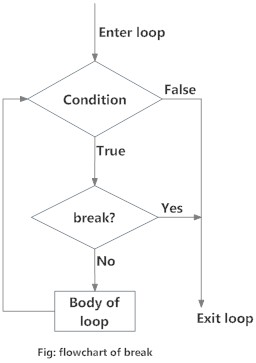
# do something

**Loop Control**

We were introduced to control flow in the previous tabs with if and else statements. Loops, breaks and continues are all a part of control flow as well. Control flow is the cornerstone of most programming languages.

When you want finer control over your loops, use the following statements to do so.

**Break**



The *break* statement exits the current loop prematurely, resuming execution at the first post-loop statement, just like the traditional *break* found in C or JavaScript.

The most common use for the *break* is when some external condition is triggered, requiring a hasty exit from a loop. The *break*statement can be used in both *while* and *for* loop. When loops are nested, a *break* will only exit from the innermost loop.

for val in "string":

if val == "i":

break

print val

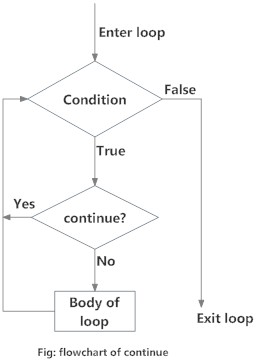
The result of the sample above would be:

s

t

r

**Continue**



The *continue* statement returns the control to the beginning of the loop. The *continue* statement rejects -- or ***skips*** -- all the remaining statements in the current iteration of the loop, and continues normal execution at the top of the loop. The *continue*statement is very useful when you want to skip one or more loop iterations, but keep looping to the end.

for val in "string":

if val == "i":

continue

print val

In this case, the result should be:

s

t

r

n

g

**Pass**

The *pass* statement is used when a statement is required syntactically but you do not want any command or code to execute.

class EmptyClass:

pass

for val in my\_string:

pass

The *pass* statement is a null operation; nothing happens when it executes. The *pass* is almost never seen in final production, but can be useful in places where your code has not been completed yet.

**Else**

There are certain conditions that we give for every loop that we have, but what if the condition was not met and we still would like to do something if that happens? We can then use else. Yes, that is right, else in a loop.

x = 3

y = x

while y > 0:

print y

y = y - 1

else:

print "Final else statement"

The output would be:

3

2

1

Final else statement

Note that this *else* code section is only executed if the *while* loop runs normally and its conditional is false (whether we never entered the *while* loop, or we did but eventually the conditional changed from true to false). If instead our *while* loop is exited prematurely because of a *break* or *return* statement, then the *else* code section will never be executed.

x = 3

y = x

while y > 0:

print y

y = y - 1

if y == 0:

break

else:

print "Final else statement"

Because of the break, the above code will output the following:

3

2

1

Abstraction is used in flask

You are pulling things from the computer

Servers generally require their own file for the server side code

INDEX.HTML SHOULD BE INSIDE OF TEMPLATES!!!

From flask import Flask, render\_template

App = Flask(\_\_name\_\_)

#/

@app.route(“/”) This is a Python Decorator, hence why not needed indentation

def index ():

return “hello world”

@app.route(“/hello”) This is a Python Decorator, hence why not needed indentation

def index ():

return “hello world”

App.run (debug=True)

You only need to refresh a sever to check on it if you broke the code and cause an error

It knows to import the object Flask because it is capitalized

A client is also required

An initial request will have nothing in it, the server still needs functionality added

Form action =”route of the url”

Form action =”/hello”

TWO FUNCTION NAMES CAN NEVER BE THE SAME

You can pass data to a render template function

Return render\_template (“index.html” name=