**Strings**

Strings are any sequence of characters (letters, numerals, ~($/}\#, etc.) enclosed in single or double quotes. You can display a string like this:

print "this is a sample string"

Copy

**Printing Strings using Variables**

There are multiple ways that you can print a string containing data from variables.

The first is by adding a comma after the string, followed by the variable. Note that the comma is *outside* the closing quotation mark of the string. Print inserts a space between elements separated by a comma.

name = "Zen"

print "My name is", name

Copy

The second is by concatenating the contents into a new string, with the help of **+**.

name = "Zen"

print "My name is " + name

Copy

There is one other difference between concatenating using a plus and using a comma, can you find out what it is?

**Hint:** try concatenating a string with an integer using each method.

Lastly, you can use curly brackets - **{}** - and the string **.format()** method to inject variables into your string - this is known as **string interpolation**.

first\_name = "Zen"

last\_name = "Coder"

print "My name is {} {}".format(first\_name, last\_name)

Copy

Above the string "Zen" is inserted where the first curly bracket is and the string "last\_name" where the second curly bracket is. There should be a corresponding number of curly brackets and arguments passed to the .format() function

As you read other people's code, you may see a different method of string interpolation. It is a lesser-used and soon-to-be deprecated method that you should know about, but will not need to use.

hw = "hello %s" % 'world'

print hw

# the output would be:

# hello world

Copy

There are several variations and tricks with each method, which have changed according to the Python version you are using. The developers of Python have yet to decide on how best to implement string interpolation for Python. Exciting stuff. Stay tuned. Python 3.6 is set to implement a new string interpolation method.

**Built-In String Methods**

String methods are functions that we can run on a string. We already showed you one above, the .format() method. Here's how to use these methods:

x = "Hello World"

print x.upper()

# output:

"HELLO WORLD"

Copy

**The following is a list of commonly used string methods:**

* string.count(substring): returns number of occurrences of substring in string.
* string.endswith(substring): returns a boolean based upon whether the last characters of string match substring.
* string.find(substring): returns the index of the start of the first occurrence of substring within string.
* string.isalnum(): returns boolean depending on whether the string's length is > 0 and all characters are alphanumeric (letters and numbers only). Strings that include spaces and punctuation will return False for this method. Similar methods include .isalpha(), .isdigit(), .islower(), .isupper(), and so on. All return booleans.
* string.join(list): returns a string that is all strings within our set (in this case a list) concatenated.
* string.split(): returns a list of values where string is split at the given character. Without a parameter the default split is at every space.

It's important to know that there are built-in methods for every data type, and to have a general idea of what they can do. Try experimenting with them in the shell to see what they can do. Don't spend time trying to memorize them, though. You can always look up whatever you need to use.

Click [here](https://docs.python.org/2.6/library/string.html) for a list of Python's built-in string methods.

**Lists**

A **list**, also known as an array in other programming languages, is a data type that allows you to hold groups of values. Think of a list like a dresser with multiple drawers in which each drawer stores some information. Lists are created with values inside of square brackets **[]**, where each value is separated by a comma. After a list is created, it can still be updated by adding values and/or by deleting values. An empty list is simply **[ ]**.

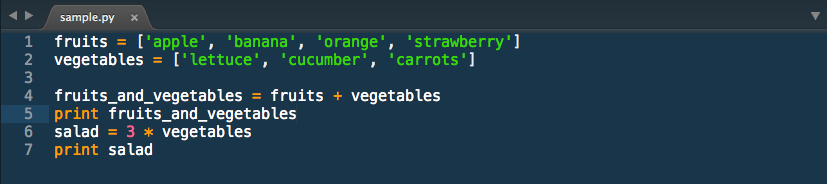
ninjas = ['Rozen', 'KB', 'Oliver']

my\_list = ['4', ['list', 'in', 'a', 'list'], 987]

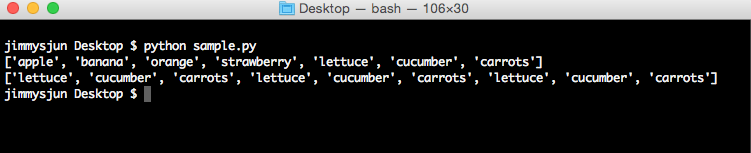
empty\_list = []

Copy

In Python, the elements of a list do not have to be of the same data type. A list can be a mixture of any Python data types, including, tuples, strings, numeric, and even a list itself (a list within a list). An example:



And if we run the code, the output would look like:



**Accessing Values**

Back to the dresser analogy, imagine that each drawer is numbered starting with 0. Say the first drawer( index of 0) has 'documents' inside, the second drawer (index 1) has 'envelopes' inside, and so on. Each drawer holds a number, also known as the index (which serves as the unique address that points to each of our items inside the drawer). You can access the items in the drawer like below:

drawer = ['documents', 'envelopes', 'pens']

#access the drawer with index of 0 and print value

print drawer[0] #prints documents

#access the drawer with index of 1 and print value

print drawer[1] #prints envelopes

#access the drawer with index of 2 and print value

print drawer[2] #prints pens

Copy

**Manipulating Lists**

Here's a useful example of a method that we will use to manipulate lists:

**<list>.append(<new\_element>)**

Appends a new item onto the end of the given list. You can pass any data type into this function.

x = [1,2,3,4,5]

x.append(99)

print x

#the output would be [1,2,3,4,5,99]

Copy

It's important to know that Python uses [ ] characters to return a copy of the list, constrained to the specified indices. This can be thought of as behaving like the slice function in JavaScript. The starting index and ending index should be separated by the "**:"**character.

x = [99,4,2,5,-3];

print x[:]

#the output would be [99,4,2,5,-3]

print x[1:]

#the output would be [4,2,5,-3];

print x[:4]

#the output would be [99,4,2,5]

print x[2:4]

#the output would be [2,5];

Copy

For information on other available list methods, [read the docs.](https://docs.python.org/2/tutorial/datastructures.html)

**List Built-in Functions**

Below is an example of a built-in function that deals with lists. The following functions can also be applied to all sequences, including tuples and strings. What do we mean when we say sequence? Think of a sequence as anything over which we can iterate. Here's one commonly used sequence function:

**len(sequence)**: Returns the number of items in a sequence.

my\_list = [1, 'Zen', 'hi']

print len(my\_list)

# output

3

Copy

**Some built-in functions for sequences:**

* enumerate(sequence) used in a for loop context to return two-item-tuple for each item in the list indicating the index followed by the value at that index.
* map(function, sequence) applies the function to every item in the sequence you pass in. Returns a list of the results.
* min(sequence) returns the lowest value in a sequence.
* sorted(sequence) returns a sorted sequence

There are a few other useful built-in functions. Find them [here](https://docs.python.org/2/library/functions.html).

**List Built-in Methods**

Below is an example of a built-in list method. These methods are specific to lists versus other sets, much like the string methods shown in the previous tab.

**list.append(value)**

my\_list = [1,5,2,8,4]

my\_list.append(7)

print my\_list

# output:

# [1,5,2,8,4,7]

Copy

**The following are some commonly used list methods:**

* list.extend(list2) adds all values from a second sequence to the end of the original sequence.
* list.pop(index) remove a value at given position. if no parameter is passed, defaults to final value in the list.
* list.index(value) returns the index position in a list for the given parameter.

These are just some of the things you can do to manipulate or extract information from a list. Click [here](http://www.linuxtopia.org/online_books/programming_books/python_programming/python_ch14s07.html) to learn more about other built-in functions you can use with a *list*.

# Conditional Expressions

Conditional statements or expressions in Python can be done using **if (and else)** just like in other programming languages. We use these conditional statements with logic operators to control the flow of our programs.

# if statement:

if <condition>:

# do something

# if-else statement:

elif <condition>:

# do something

else:

# do this instead

Copy

Say, for example, you were driving home and there was some construction on the road in front of you. You notice a detour sign and decide to take that way back instead. Although it was practically a subconscious decision, this illustrates how we use control flow and conditionals in everyday life to determine what we would do based on certain conditions. Our if-else statement would look like this:

***If there is construction******{******use detour*** ***}******else******{******take the normal route******}***

Here's another example but now written out in python code:

age = 15

if age >= 18:

print 'Legal age'

else:

print 'You are so young!'

Copy

The if and if-else statements in Python are straightforward and are very much like the if statements in other languages. The only difference with Python's if statement is, when you have another condition, you write it using **elif.**

if age >= 18:

print 'Legal age'

elif age == 17:

print 'You are seventeen.'

else:

print 'You are so young!'

Copy

**elif**is just like **else if** or **elsif**from other languages.

## Comparison and Logic Operators

Here is a table of the comparison operators you can use in your Python programs.

|  |  |  |
| --- | --- | --- |
| **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. |

\*Note: != can also be written <>, but this is an obsolete usage kept for backwards compatibility only. New code should always use !=.  Documentation can be[found here](https://docs.python.org/2/library/stdtypes.html).

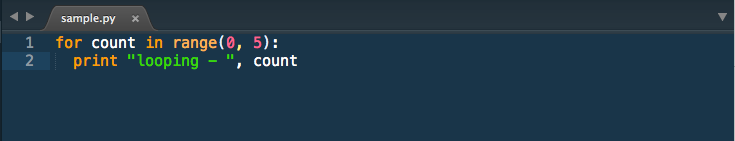
# Loops

Imagine that you are in 1st grade and you got in trouble in class for talking too much (it happened to me a lot of times). Your teacher asks you to write "I will not talk in class" 1,000 times. Yikes! If you had learned to program in kindergarten, you might have thought to write a program that uses a loop to do it for you!

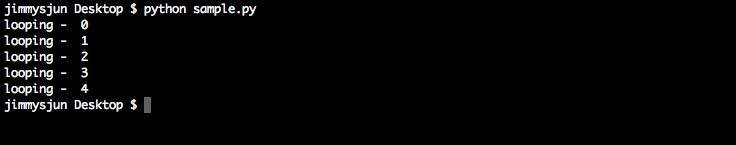
In Python, like many other programming languages, loops are the way of executing a set of code repeatedly for a certain amount of iterations or until we've reached a specific condition. This is because computers are great at **doing things over and over again**. This could be used for something as simple as a math program that counts from 1 to 1,000,000 or iterating through the items within a list! In this section, we will be talking about the **for**and **while**loops in Python. In essence, anything you can do with one loop type, you can do with the other, but let's see how they are different.

## For Loop

We use the for loop **when we know how many times we have to repeat our code**. You will mostly be using for loops in your programs, particularly in Python. A for loop looks like this:



with an output that looks like this:



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:

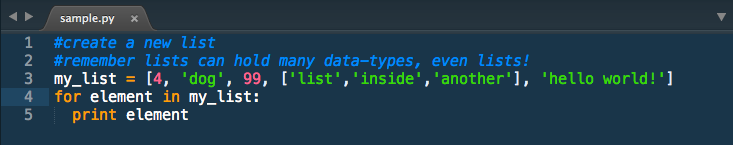
for <counter> in <sequence or range>:

# do something

Copy

### Looping Through a List

Often you'll find yourself wanting to loop through a list.



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']

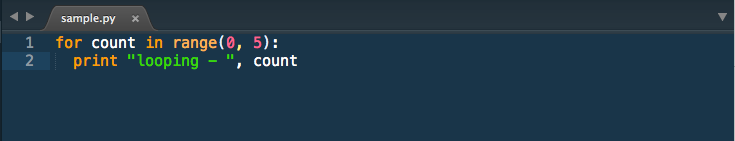
hello world!

Copy

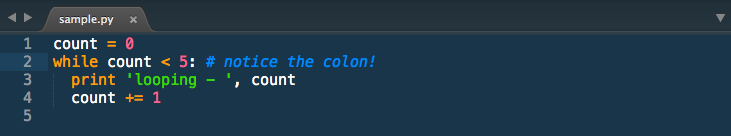
## While Loops

**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

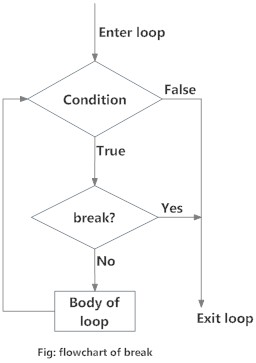
Copy

## 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 breakstatement 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

Copy

The result of the sample above would be:

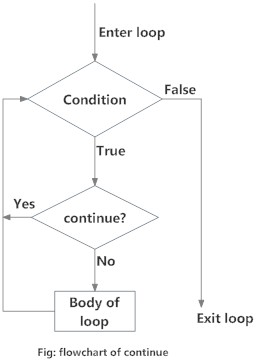
s

t

r

Copy

#### 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 continuestatement 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

Copy

In this case, the result should be:

s

t

r

n

g

Copy

#### 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

Copy

for val in my\_string:

pass

Copy

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"

Copy

The output would be:

3

2

1

Final else statement

Copy

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"

Copy

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

3

2

1

# Functions

A **function** is a **named block of code that we can execute to perform a specific task**. More simply, a function is a list of instructions that you can run at any time. If you find something that you seem to be using over and over again, it might be best to have a way to streamline the process. A function can optionally take in parameters, perform a series of instructions, and optionally return something afterwards. Here's an example:

def add(a,b):

x = a + b

return x

# the return value gets assigned to the "result" variable

result = add(3,5)

print result # this should print 8

Copy

Think of the function as a factory. If we were building a new car we would:

1. Acquire raw materials (variables) needed for creating a car.
2. Send the raw materials(invoke and pass arguments) to a car manufacturing plant (function)
3. Do something (process) with the raw materials(parameters)
4. Drive the car (function's return value)



The factory has all the instructions to build a new car and will perform all the tasks. When you want a new car, all you have to do is call the factory to request a new car.

The advantages of using functions are:

* Reducing the duplication of code
* Breaking down complex problems into simpler pieces
* Improving clarity of code

## Syntax

Pay attention to a few details. The def keyword signifies the declaration of a function. This indicates that the following code is a function and assigns a name to that function, so we can **call** it later. Parameters are information we **input into a function**, and appear inside the parenthesis that follow the function name.

Here's a basic example of a function:

# we've named the function 'add' and we give it two parameters(inputs to the function)

def add(a,b):

x = a + b

return x # we return something (more on this later)

Copy

We have declared a function with the def keyword, named it add, and it takes two inputs (parameters). An important thing to know is that the above code does not actually invoke the function; it just declares it. Once you've defined your function, we can execute it by **invoking** or **calling it** using **()** after the function name.

print add(3,5) #prints 8

Copy

Once invoked, a function can give us an output. Some functions take an input and some functions don't give us an output. Even if no output is produced, the code inside the function can alter the program - this is called a side effect. Based upon what we learned above, a function that doesn't return anything would produce no output!

## Function Parameters

We define the input of functions using **parameters.** Like we've seen before, some functions do not have to take parameters. However, functions can optionally have one or more parameters.

We've defined the say\_hi function with one parameter called name

# this function has one parameter(input)

def say\_hi(name):

print "Hi, " + name

Copy

Now, we can invoke this function by calling its name and passing in the correct number of arguments:

# invoking the function passing in one argument

say\_hi('Michael')

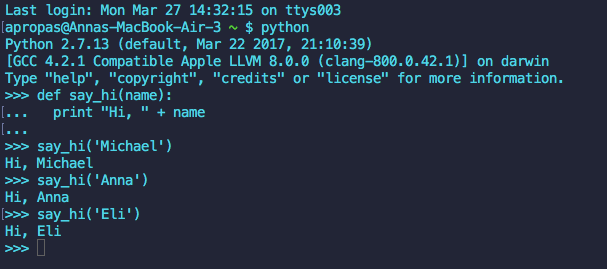
say\_hi('Anna')

say\_hi('Eli')

Copy

Wait, but what's the difference between a parameter and an argument? These two words get mixed up a lot in programming. In this example **'name' is a parameter** while **"Michael", "Andrew", and "Jay", are arguments**. We **define parameters.** We **pass in arguments** into functions.

Here's the output:



## Returning Values

So far none of our functions had any value that we could hold onto. In many cases, we would want our function to **return** some sort of value that we can use later in our program. The following concept is critical in understanding how to use functions correctly in your code:

It is very important to remember the following: **a functional call is equal to whatever that function returns**. This might not make sense until we see it in action.

Let's modify our original say\_hi function and observe the differences:

def say\_hi():

return "Hi"

greeting = say\_hi() #the returned value from say\_hi function gets assigned to the 'greeting' variable

print greeting # this will output 'Hi'

Copy

Returning a value from a function allows us to store that value in a variable. In this example, we invoked the say\_hi function and set it to the greeting variable. When we print greeting we see that it contains the returned value of the say\_hi function - "Hi'

Going back to our add function, recall that it takes two parameters and returns the sum of the parameters.

def add(a, b):

x = a + b

return x

sum1 = add(4,6)

sum2 = add(1,4)

sum3 = sum1 + sum2

Copy

What do you think the values of sum1, sum2, and sum3 would be?

If you guessed 10, 5, and 15, respectively, good job! sum1 was set to the return value of the add function invoked with 4 and 6 as arguments. Similarly, sum2 was set to the return value of invoking add with 1 and 4. The variable sum3 contains the sum of sum1 and sum2 which is 15. Storing these return values in variables allows us to use the results of our functions throughout the rest of our program.

In our examples you may have noticed that our functions were returning values of different data types. Functions can return any of the data types - strings, numbers, lists, tuples, dictionaries and even other functions!