Contents

1 Introduction........3
1.1 Hello World!........3
1.2 Who am I........3
1.3 Who we are........3
1.4 What makes Python Awesome?........4
2 Fundamentals of Python Programming........8
2.1 Literals........8
2.2 Numbers........9
2.3 Operators........9
2.4 Strings........13
2.5 Data Type........18
3 Data structure in Python........22
3.1 List........22
3.2 Tuple........27
3.3 Dictionary........30
3.4 Set........36
4 Program Design in Python........39
4.1 Decision Structure........39
4.2 Functions........44
4.3 Functional Programming........48
4.4 Modules........49
4.5 Object Oriented Programming........51
5 Advanced Topics........55
5.1 File Input/Output........55
5.2 Exceptions Handling........58
5.3 Time........58
5.4 Regular Expression........60
5.5 Compare Sequences........63
5.6 Problem Set: Time for a Challenge!........63

1 Introduction

1.4 What makes Python Awesome?

There are some reasons to describe why I love Python:

1.4.1 It is FREE!!

I've personally have not heard a better reason to use something. Python is a totally free language to download, use and play with, that's because a bunch of crazy volunteers devote their time to improving the language (much like Wikipedia).

- Free License
- Free learning resources
- Great Communities
- Free repository (PyPi)

1.4.2 Easy to learn, Fast production, Easy Maintenance!!

- Python code is often 3-5 times shorter than Java, and 5-10 times shorter than C++.
- Python code is incredibly readable and often close to the English language.

It is EASY: Let's do a COMPARISION!

A **for** loop over a list in Python:

```
items = [1,2,3,4,5]
for x in items:
    print x
```

And same thing in Ruby:

```
items = [1,2,3,4,5]
items.each do |i|
     puts i
end
```

Another Comparison:

A **while** loop in Python...

```
x = 1
while x < 5:
      print "The number is:", x
      x += 1
```

And the same code in php:

```
<?php
$x=1;
while($x < 5)
{
      echo "The number is: $x <br>";
      x++;
}
?>
```

I love syntactic clarity in Python. It's basically executable pseudo code!

1.4.3 What we believe in

Everyone should learn programming - because programming teaches you how to think and then, makes you able to show the others how you think!

So we need a language that is easy to learn for everyone. That is where Python comes in.

- Python is easy to learn on your own. Good for self-study.
- Similar to the English Language! This makes it easy to remember commands and also makes it easy to understand what you are doing.
- It is your **Stepping Stone** to the world of programming!

1.4.4 Professionals use it!

Remember *easy maintenance and fast production + free license =*

- Official language at Google
- YAHOO used Python for Yahoo Answer and other projects!
- Disney
- Nokia
- IBM
- Games (Civilization, Battelfield)
- Genomics

1.4.5 Kids can use it!

Children from 9 years old are able to use it! Hundreds of school around the worlds use python for teaching Coding to the kids.

1.4.6 Hardware Programming: Hello Raspberry Pi!

Raspberry Pi is a card-sized, inexpensive microcomputer that is being used for a surprising range of exciting do-it-yourself stuff, such as: robots, remote-controlled cars, and video game consoles. With Python as its main programming language, Raspberry Pi is being used by kids to build radios, cameras, arcade machines, and pet feeders!

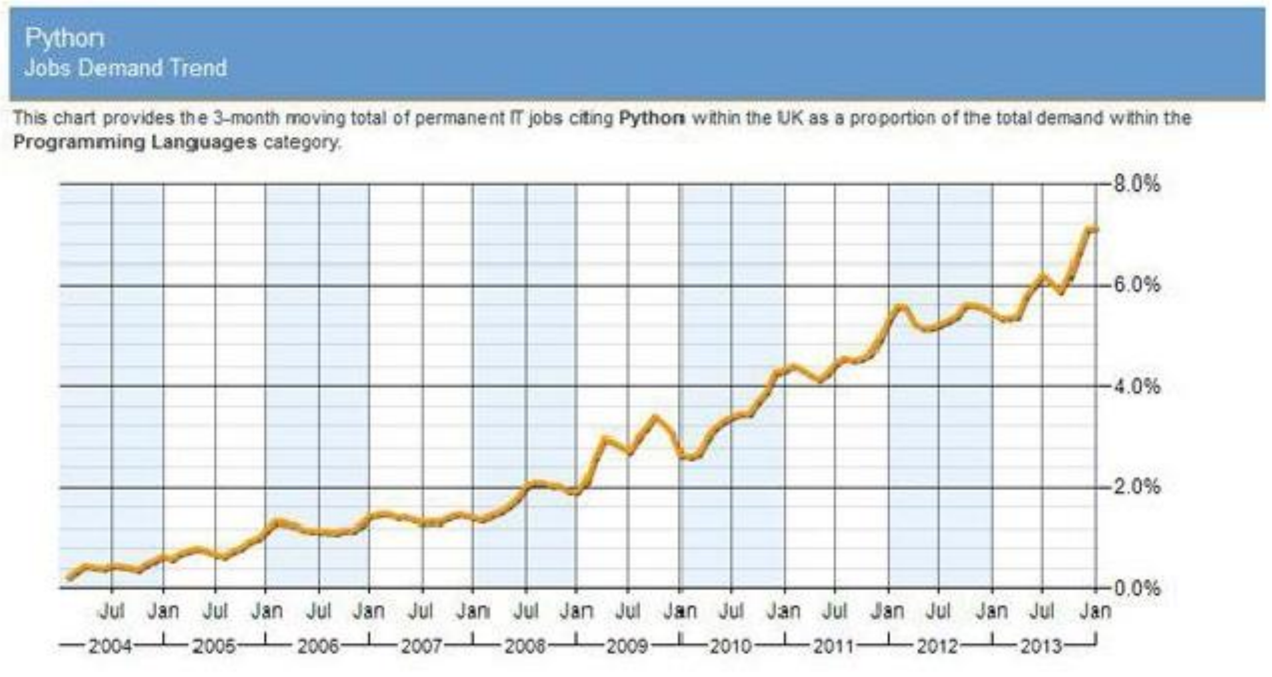
1.4.7 Money Money Money

Companies such as Google, Yahoo!, Disney, Nokia, and IBM all use Python. In fact, among programming languages, Python had the largest year-on-year job demand growth — at 19% — as of March 2013.

Notably, the overall hiring demand for IT professionals dipped year over year by 5% as of January 2014, except for Python programmers which increased by 8.7%.

In New York, Python developers ranked #8 of the most in-demand tech workers, making an average of $106k per year.

On the other side of the Atlantic, Python programmers also enjoy a rising demand for their skills as shown by the following graph.



1.4.8 It's Online!

Web development is still a booming economic prospect for programmers. So Python strongly supports web development through many different web frameworks such as **Django, CherryPy, Tornado, Flask,** ....

Django — the popular open source web application framework written in Python — is the foundation of such sites as **Pinterest, The New York Times, The Guardian, Bit Bucket**, and **Instagram.** Django is a complete framework that takes the complexity out of web development, while still giving you control over as much as you want. As an open-source framework, all the information you need to get started can be found at DjangoProject.com.

1.4.9 The Father of Python!

Python is a programming language created by Guido Van Rossum in the early 90's. It is now one of the most popular languages in existence.

2 Fundamentals of Python Programming

At the first step of learning Python, l et’s talk about some fundamental topics in Python Programming: Literals, Numbers, Operators, Strings, and Data Types.

2.1 Literals

Lets' get definition from Wikipedia)

“ In computer science, a literal is a notation for representing a fixed value in source code. Almost all programming languages have notations for atomic values such as integers, floating-point numbers, and strings, and usually for Booleans and characters; In contrast to literals, variables or constants are symbols that can take on one of a class of fixed values, the constant being constrained not to change”.

In the following code you can see examples of different types of literals in Python...

```
#Integer Literal
print 42343

#Long Integer
print 2342342L

#Floating Point
print 343.3234

#String Literal
print "Hi This is Good"
print 'Thank you'

#Binary Literal
print 0b101 * 0b10

#Hexadecimal Literal
print 0x100 / 0x2

#Octal Literal
print 0400 / 02
```

To **print** the result of an expression in python, we use a `print()` function. In the above example we print many different numerical values. At the same time you can see some simple mathematical operations.

What is #?

Whatever you write after **#** is called a **comment** and that means it will never be executed. We use comments to add explanation to our code, then we can be sure to remember what we have done - or enable other people to understand our code.

String Delimiter

About strings, we will discuss more, but for now, you can use either single or double quotes for string delimiters. The following string literals are valid:

```
"Hi" or 'Hi'
```

2.2 Numbers

In Python you have everything a calculator does - and more! Let's try the following code!

```
# Print basic integers
print 5
print 5.5
print -5
# Combine them with normal mathematic operators
print 1+1
print 8-1
print 10*2
print (-4) + 4
# Decimal place numbers work just fine
print 2.5 * 2
# Sometimes the operators are a little different, however
# 2 to the power of 4
print 2**4
print 35 / 2
# Calculate the remainder of 7 divided by 3
# Can you explain what the '%' operator does?
print 7 % 3
```

2.3 Operators

Operators manipulate your data and generate new data! Any programming language regardless of its syntax, follows common operators such as arithmetic, logical, or Boolean operators. Let's get familiar with these operators.

2.3.1 Arithmetic Operators

```
print 5 + 5
print 5 * 5
print 5 - 5
#Integer Division
print 5 / 2
print 5 / 2.0
#Floor Division
print 5 // 2
print 5 // 2.0
#Modulus Operator
print 7 % 2
print 2 ** 4
```

2.3.2 Assignment Operators

In Python you can declare and initialize a variable without specifying any data type. Remember that Python is a dynamic language. We will talk more about variable later.

```
x = 4
print 'x = 4',x
x += 4
print 'x += 4 =>', x
x -= 4
print 'x -= 4 =>',x
x *= 4
print 'x *= 4 =>',x
x /= 4
print 'x /= 4 =>',x
x %= 3
print 'x %= 3 =>',x
x = 2
x **= 4
print 'x **= 4 =>',x
```

2.3.3 Boolean Operators

In computer science, a **Boolean expression** is an expression in a programming language that produces a Boolean value when evaluated, i.e. either `True` or `False.`

A Boolean expression may be composed of a combination of the Boolean constants `True` or `False`, Boolean-typed variables, Boolean-valued operators, and Boolean-valued functions.

Let's see some Boolean expressions:

```
print 3 > 2
print 3 < 1
print True
print False
```

2.3.4 Comparison Operators

Comparative operators in Python:

```
print 5 == 5
print 5 != 5
print 5 <> 5
print 5 > 5
print 5 < 5
print 5 >= 5
print 5 <= 5
```

2.3.5 Logical Operators

Using a few logical operators you can create complex Boolean expressions:

```
print 4 > 3 and 2 > 1
print 3 < 2 and 6 > 3
print 3 != 2 or 3 > 9
print not 4 == 4
print 4 != 4
print not 5 > 2
```

2.3.6 Bitwise Operators

In digital computer programming, a bitwise operation operates on one or more bit patterns or binary numerals at the level of their individual bits. It is a fast, primitive action directly supported by the processor, and is used to manipulate values for comparisons and calculations.

```
#Binary And
print '5 & 4 =', 5 & 4
#Binary Or
print '4 | 1 =', 4 | 1
#Binary XOR
print '5 ^ 3 =',5 ^ 3
#Binary Complement
print '~5 =',~5
#Binary Left Shift
print '4 << 1 =', 4 << 1
#Binary Right Shift
```

```
print '4 >> 1 =', 4 >> 1
```

2.3.7 Membership & Identity Checker

We have two useful operators in Python that save both a lot of pain! First off is the **in** membership checker. It gets an element and a collection, then checks whether that element exists within the collection or not.

```
print 5 in [4, 5, 3, 4, 1]
# Read: Is the number 5 'IN' the following array, [4, 5, 3, 4, 1]
print 5 not in [4, 5, 3, 4, 1]
print 'and' in 'This is a long message'
```

Another useful operator is **is**. This is more low-level! It accepts two objects (or variables) then evaluates to true if the variables on either side of the operator point to the same object and false otherwise.

```
x = 5
z = x
y = 4
print 'x is y', x is y
print 'x is z', x is z
print 'x is not y', x is not y
```

2.3.8 Operator Precedence!

Who's first!

Ok, let's finish operators with the concept of precedence!

```
1.     **
2.     + - (Unary plus and minus)
3.     / % //
4.     ~ + - (Addition, Subtraction)
5.     >> <<
6.     &
7.     ^ |
8.     <= < > >=
9.     <> == !=
10.    = %= /= //= -= += **=
11.    is, is not
12.    not, or, and
```

Objects

Up to this point you have heard the term **variables** many times! However, it would be more correct to refer to everything in Python as an **object**! If you are familiar with object oriented programming, this shouldn't be any problem for you. If not, look over the following examples:

We have a string variable `message` that holds a textual message.

message = 'This is a text'

If `message` is just a variable you don't expect anything else from the variable except for it to hold a value. Whenever you need that value you can refer to this variable name. Though what if you need to get the uppercase version of the content in `message`. Wouldn't it be great if variable could do this for us? Well, it can:

```
print message.upper()
```

Look at the **dot operator** '`.`'. I have used after `message` and then `.upper()`. I am calling one of the many functionalities of the string OBJECT (not variable) to make it upper case.

So if you have many string objects like `message`, each one of them has its own inbuilt functionality and properties. That's why we don't call it a variable! Everything in Python is an object! So always expect some additional functionality when using **dot** '`.`' after a variable; **Oh sorry, an object!**

2.4 Strings

You have seen numbers and simple examples of the string data type so far! We used strings to show our sequences. Strings in Python are sequences of characters. You create a string literal by enclosing the characters in single `'`, double `"` or triple quotes `'''` or `"""`.

```
text = "Computer science is no more about computers than astronomy is
about telescopes."
print "Dijkstra:",text

text = 'Computer science is no more about computers than astronomy is
about telescopes.'
print "Dijkstra:",text

# Triple quotes have the added benefit of allowing our code to span
multiple lines.
text = '''Providing better computer science education in public schools
to kids, and encouraging girls to participate, is the only way to
rewrite stereotypes about tech and really break open the old-boys'
club.'''
print "Dijkstra:",text
```

2.4.1 Indexing

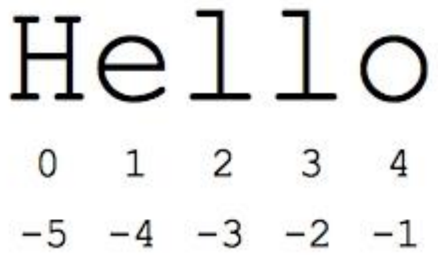
You can access parts of a string using the indexing operator `yourString[i]`. Indexing begins at zero, so `yourString[0]` returns the first character in the string, `yourString[1]` returns the second, and so on.

```
letters = 'abcdefghijklmnopqrstuvwxyz'
```

```
print letters[0]
print letters[1]
print letters[4]
print letters[7]
```

2.4.2 Negative Indexing

It's cool! Negative indexing is simple, lovely, and saves you a lot of time!



So try it!

```
letters =  'abcdefghijklmnopqrstuvwxyz'
print letters[-1]
print letters[-2]
print letters[-4]
print letters[-7]
```

2.4.3 Slicing

Biologists find the **slicing** functionality in Python very useful. Slicing provides you with a simple approach to return any portion of a sequence.

The general form of slicing is `seq[start:stop:step]`

Let's try...

```
letters = 'abcdefghijklmnopqrstuvwxyz'
print letters
#Print sequence from 5th base until 15th
print letters[5:15]
#Print sequence from begining base until 15th
print letters[:15]
#Print sequence from 5th until the end
print letters[5:]
```

Try Negative Indexes

Everything is like positive indexing except the direction is right to left! **Remember** the `start` value must be less than `stop`.

```
letters = 'abcdefghijklmnopqrstuvwxyz'
print letters
print letters[-15:-55] #Does not work
print letters[:-5]
print letters[-15:]
```

Step

Try 'step' to return the data you need.

```
letters = 'abcdefghijklmnopqrstuvwxyz'
print letters
#From begining until the end but only every 3rd char!
print letters[::3]
print letters[::1]
print letters[5:15:3]
```

What about negative steps? Like `letters[::-1]` What do you think will happen?

```
letters = 'abcdefghijklmnopqrstuvwxyz'
#Try this letters[::-1]
```

2.4.4 String method

There are many useful functions for strings that are already designed and embedded into Python. Normally (not always) the way we can have access to those functions is with the **dot** operator:

<data>.<function name>

Let's get familiar with some string functions:

Example 1:

```
text = "Object-oriented programming is an exceptionally bad idea which
could only have originated in California."
print text.upper()
print text.lower()
```

Example 2:

```
#I like this next function it always comes in handy
```

```
print text.capitalize() #Capitalizes first letter of string
```

Example 3:

```
text = "Object-oriented programming is an exceptionally bad idea which
could only have originated in California."
print len(text)
print text.startswith("This")
print text.endswith("This")
```

Example 4:

```
#There is a nice counter for the string data type and this also works
with lists
print text.count("i")
print text.count("is")
#You can count in a substring, e.g. from index 0 until index 5
print text.count("is", 0,   5)
```

Example 5:

```
#remove dummy characters from BEGINING and END of a string
print "text    ".strip()
print "text+++".strip("+")
#You may want to trim all leading and  trailing whitespace in string
print "  text    ".strip()
#You can add other characters to be trimed
print "  text   +++||||".strip("|+ ")
```

Example 6:

```
text = "Object-oriented programming is an exceptionally bad idea which
could only have originated in California."
#check if a specific word exists in the text or not
print "is" in text
```

Example 7:

```
#Find function determine if another str occurs in a string
#returns index if found and -1 otherwise
print text.find("is")
print text.find("is", 5)
print text.find("isss")
```

Example 8:

```
#Splits string according to delimiter str (space if not provided) and
returns list of substrings;
print text.split() #Split based on whitespace
print text.split(",")
print text.split("is")
```

2.4.5 String Formatting

When you are dealing with strings you may want to print the same string that differs only in one or two places. Consider following example:

```
Tom purchased Picca at 8:10PM


Jack purchased Sandwich at 9:10PM


John purchased Water at 9:30PM
```

As you see there is a common parts of each sentence:

```
{name} purchased {food} at {time}
```

We can create a string template and format it with real data that comes from a database. This could save us a lot of time and provide better structure for our code. Let's see what available approaches to do this are.

Old Fashioned! C Style

Good news for C programmer, just do what you have done with `printf`.

```
# Zero left
print 'Now is %02d:%02d.' % (6, 30)

# Real (number after the decimal point)
print 'Precentage: %.1f%%, Exponential:%.2e' % (5.333, 0.00314)

# Octal and hexadecimal
print 'Decimal: %d, Octal: %o, Hexadecimal: %x' % (10, 10, 10)
```

Operator % is used to make string interpolation. Interpolation is more efficient in memory usage than conventional concatenation.

Symbols used for interpolation:

- % S: string
- % D: integer
- % O: octal
- % X: hexadecimal
- % F: real
- % E: real exponential
- %%: Percent sign

Symbols can be used to display numbers in various formats.

2.4.6 Modern Approach

From Python version 2.6, there is an alternative to using the interpolation operator %, this method is called: `format()` .

Example:

```
db_factors = [('Pizza', '08:10PM', 'Tom'), ('Sandwich', '09:00PM',
'Jack')]

# Parameters are defined in order
msg = '{0} purchased by {1} at {2}'

for food, buyer, time in db_factors:
    print(msg.format(food, buyer, time))

# Parameters are defined by name
msg = '{food} purchased by {buyer} at {hour:02d}:{minutes:02d}'

print msg.format(food='Pizza', buyer = 'Tom', hour=8, minutes=10)

# Builtin funcation format()
print 'Pi =', format(3.14159, '.3e')
```

2.5 Data Type

If you have ever used C or C++ you must know what `typeof()` does! If not, remember that everything in a programming language has a type!

First let me ask my dear friend to explain the meaning of type. Dear Mr. Wikipedia, please let us know what `type` is:

“ In computer science and computer programming, a data type, or simply type, is a classification for identifying one of various types of data, such as real, integer or Boolean, that determines the possible values for that type; the operations that can be done on values of that type; the meaning of the data; and the way values of that type can be stored”.

This Description of **type** is heavily dependent on the physical structure of allocated memory for that type. For example in most programming languages we have different data types for integer values. For example in C, we have **byte** where only 1 byte will be allocated to a variable of that kind, so 1 byte is 8 bits and we can present 256 different numbers! While for another data type **short** 2 bytes will be allocated, meaning 2 to the power of 16 different numbers.

Good News! Python is a dynamically typed language, meaning it will detect your variables and expression data type in runtime. But many times you have to check the type by yourself before doing some relevant operations. This can be done with the **type** operator.

```
print type(5)
print type(5.0)
print type(5) == int
print type(5) == float
print type('but I want to be an integer!')
print type(5/2.0)
```

2.5.1 Freedom of Type

Everything has a type! It doesn't matter if you are in a static or a dynamic language! As long as you are a programmer, you must always ask this question! Even when you go to a shop and the shopkeeper says, "Its $20", you have to ask, "Sorry, is this in integer, long, or float data type? Oh, what about double!!"

Python is a dynamic language and types are declared at runtime. What are the available types in Python?

Everything is done at runtime, but there are several simple types of pre-defined data in Python, such as:

- Numbers (integers, reals, complex, ...)
- Text (or strings)

Furthermore, there are other types that acts as collections. The main ones are:

- List
- Tuple
- Dictionary
- Set
- Named Tuple

- Ordered Dictionary
- Frozen Set

We will explain some of them later

How does Python manage variable declaration and initialization?

In Python variable names are references that can be changed at runtime. The most common types and routines are implemented as **built-in**s, meaning they are always available at runtime without the need to import any library.

For example:

```
x = 5
```

After this line, Python detects the proper data type (which is integer) and also reserves a memory location where it stores the integer. Also it is important to remember a variable can be assigned to any data types at runtime

```
x = 5
x = 3.4
x = "Hello"
```

2.5.2 Check type

In dynamic languages we need to be able to check and verify a variable data type. This can be done by using the **type()** function

```
x = 5
print type(x)
x = 5.4
print type(x)
x = "hello"
print type(x)
print type(3*3.0)
print type(3*3)
```

2.5.3 Where is its location?

If you have used C, you will know about getting the address of allocated memory to a created variable. In Python, for any created object a unique integer **id** is created and that would be a key that maps the created object into its memory location. In **Python Manual** we have this description:

*The **identity** of an object. This is an integer (or long integer) which is guaranteed to be unique and constant for this object during its lifetime. Two objects with non-overlapping lifetimes may have the same `id()` value. (Implementation note: this is the address of the object.)*

Let's try it.

```
x = 5
y = "hi"
print id(x)
print id(y)
```

2.5.4 Mutable VS. Immutable

Remember that everything in Python is an object! In object oriented programming, so also in Python, objects can be:

- Mutable: allow the contents of a variable to change (think mutation).
- Immutable: do not allow the contents of variables to change.

For example in Python a **list** is mutable while a **string** is not! Let's create a string object and then try to change it's character at position 2.

```
s = "abcdefg"
s[2] = 'x'
```

Look at the error message, `'str' does not support item assignment on line 2`, this is the result of being immutable! But of course you can change the entire string object value with a new one, but even in that case the old object will be removed (after a while) and the a new one will be generated.

```
s = "hello"
print id(s)
s = "bye"
print id(s)
```

As you see after the second initialization, **s** is not referring to the first object, its id has changed as well as its location within the memory!

But for lists, which are mutable objects, you have this ability to change data in place:

```
i = [1,2,3]
#check current ID
print id(i)
#Try to change in place
i[0] = 7
#No error, and now check if ID is the sane or not?
print id(i)
```

3 Data structure in Python

In python there are four fundamental types of sequences to hold collections of items.

- List
- Tuple
- Dictionary
- Set

And also there are some more others collections, but they are more specific versions of the previous four:

- Named Tuple
- Ordered Dictionary
- Ordered Set
- Frozen Set

3.1 List

The first collection is List. Lists are collections of heterogeneous objects, which can be of any type, including other lists.

Mutable

The lists in Python are mutable and can be changed at any time.

Strings are Immutable Character Lists

Strings are simply a kind of character list - but immutable. So when we talk about lists, we have similar indexing, slicing, and even similar methods and functions.

3.1.1 Declaration of List

You can create empty lists like this:

```
data = []
```

Or with some primitive data:

```
data = [1,2,3,4,5]
```

Remember you can have elements of different data types in a list. Consider the following list:

```
data = [1, 2, 3, 5.6, 'What am I doing here :(']
```

Try it now...

```
data = [1, 2, 3, 5.6, 'What am I doing here :(']
print data
```

You can get the length of a list by using **len** function.

```
data = [1, 2, 3, 5.6, 'What am I doing here :(']
print len(data) #Recall: Just like a string!
```

3.1.2 Traverse a List

If you are a programmer you should be familiar with for-loops, they allow you to repeat a block of code, and especially they are good for traversing within a list. If you are not, just be patient, in next few chapter we are going to talk about them!

Traversing is just like reading!

```
colors = ['red', 'green', 'black', 'brown']
for c in colors:
    print c
```

Look at the simplicity of the **for** loop. With only a holder variable, like **c**, will iterate through the collection and pick up elements one by one then print them to the screen.

3.1.3 Range()

Range (start, stop, step)

This a useful function that generates numerical sequences.

Range accepts three arguments, then generates a list containing arithmetic progressions.

You can use range() in three different ways:

1. **range(stop)**: With one argument

A sequence of number from 0 **until** **stop** (or **stop**-1) will be generated:

```
#From 0 until 10
my_list = range(10)
print my_list
```

2. **range(start, stop)**: With two arguments

A sequence of numbers from **start** **until** **stop** will be generated:

```
#From 10 until 20
my_list = range(10, 20)
print my_list
```

3. `range(start, stop, step)`: With three arguments

A sequence of numbers from `start` **until** `stop` and with the `steps` between each number in the sequence:

```
#From 10 until 30, 2 by 2
my_list = range(10, 20, 2)
print my_list
```

We use `range()` in many different situations, especially with `for` loops. They are very useful for creating arbitrary data for playing with lists!

And you can use range for negative numbers as well.

```
print range(0, -30, -10)
```

3.1.4 Element Access

The same way as accessing characters within a string:

```
colors = ['red', 'green', 'blue', 'brown']
print colors[0]
print colors[3]
#Similarly we have negative indexes as well
print colors[-1]
```

3.1.5 Slicing

Again exactly similar to what you have seen for strings...

```
digits = range(20)
print 'digits[5:10]  ',digits[5:10]
print 'digits[5:]    ', digits[5:]
print 'digits[:15]   ',  digits[:15]
print 'digits[::2]   ', digits[::2]
print 'digits[::3]   ', digits[::3]
print 'digits[5:15:2]', digits[5:15:2]
print 'digits[5::2]  ', digits[5::2]
```

3.1.6 Modify a List

You can add, remove or replace an element within a list...

To add a new element to the end of the list you have to use `.append()`. To delete, it you must use the `.remove()` function. To remove from specific position use `.pop(index)`.

Try it:

```
bases = ['A', 'C', 'G', 'T']
print bases
bases.append('U')
print bases
bases.remove('U')
print bases
bases.pop(1)
print bases
bases.insert(1, 'C')
print bases
```

3.1.7 Existence

Similarly you can use `in` to check the existence of an element.

```
bases = ['A', 'C', 'G', 'T']
print 'U' in bases
```

3.1.8 Join

`.join` accepts a list of items and a delimiter, then converts all items to a `string` and use that delimiter to concatenate the items.

```
bases = ['A', 'C', 'G', 'T']
print '+'.join(bases)
print ' or '.join(bases)
print ''.join(bases)
```

3.1.9 Merge Two Lists

You can do different approaches. First of using **+** operator

```
list_1 = [1,2,3,4]
list_2 = [5,6,7,8]
list_1 += list_2
print list_1
```

Or you can go for another function called `.extend()`

```
list_1 = [1,2,3,4]
```

```
list_2 = [5,6,7,8]
list_1.extend(list_2)
print list_1
```

3.1.10 List Comprehension

If you want to do some modification on all elements within a list, or filter out some elements from a list, you will love list comprehension.

*Example 1: We have a list of calculated GC contents, but we want to express all of them as a percentage (meaning: each item in the list * 100)*

```
pr_list = [0.4, 0.5, 0.7, 0.6, 0.3]
print pr_list
print [ pr * 100 for pr in pr_list]
```

What's happening here?

Inside of `[ pr * 100 for pr in pr_list]` we have a simple iteration through the `pr_list`. In this iteration, all element in `pr_list` from left to right, one by one is read and assigned to the **temporary** variable `pr` and then `pr * 100` is calculated which is what we wanted! Then it will be inserted into a new list.

In general a list comprehension is like this...

```
[expression for container in list]
```

Example 2: We have a list of Celsius temperatures and want to convert them into Fahrenheit.

```
celsius = [39.2, 36.5, 37.3, 37.8]
fahrenheit = [ ((9.0/5)*x + 32) for x in celsius ]
print fahrenheit
```

Example 3: We have some text and we want to split it into its words based on the whitespaces in between. Then, to make it better, we want to strip all dummy characters from each word, and then remove all words that their length is less than 3.

```
words = "Programming is  one of the most difficult branches of applied
mathematics; the poorer mathematicians had better remain pure
mathematicians.".split(' ')
print 'Before Filter:\n', words
words = [w.strip(' ;') for w in words if len(w) > 3]
print 'After Filter:\n', words
```

As you see we can flavour list comprehensions with `if` to create a simple filter. In this case each `word` which is in temporary variable `w` will be kept if its length is more than 3, and also any space or character from beginning or end will be removed.

3.1.11 Check Existence!

There are two ways! If you want to check the existence of an item in a list:

```
progs = ['C', 'Java', 'Basic', 'Ada', 'Ada', 'Pascal', 'COBOL',
'Pascal', 'Haskell', 'Pascal']
print 'Pascal' in progs
```

If you want to get the index of the first occurrence:

```
progs = ['C', 'Java', 'Basic', 'Ada', 'Ada', 'Pascal', 'COBOL',
'Pascal', 'Haskell', 'Pascal']
print progs.index('Pascal')
```

Just remember if it does not exist Python will return an exception! There is no -1! To manage this, you must be able to handle exceptions!

```
progs = ['C', 'Java', 'Basic', 'Ada', 'Ada', 'Pascal', 'COBOL',
'Pascal', 'Haskell', 'Pascal']
try:
    index_value = progs.index('Python')
except ValueError:
    index_value = -1
print index_value
```

3.2 Tuple

3.2.1 Tuples: An Immutable List

Similar to normal lists but they are immutable: you cannot add, delete or make assignments to items.

Syntax:

```
= tuple (a, b, ..., z)
```

The parentheses are optional. But, what are the benefits of tuples over lists?? Read on and see!

3.2.2 Declaration

The same way as a lists, but instead of square brackets, you must use parentheses!

```
location = (3.12345, 101.23423)
print location
```

- A tuple with two elements is called a binary tuple
- A tuple with three elements called a ternary tuple
- In the same way a tuple with N elements called n-ary tuple

Immutable

Yes, tuples are immutable, and you can't change them! Similar to Strings.

3.2.3 Element Access

Exactly in the same way as lists!

```
data_point = (34, 45, 12, 'actived')
print data_point[2]
print data_point[-1]
```

3.2.4 Slicing

Again similar to list...

```
data_point = (34, 45, 12, 'actived')
print data_point[0:2]
print data_point[1:4]
print data_point[::2]
```

3.2.5 Unpack

The term **unpack** has a long story in Python. Here it means unpack a tuples elements into different variables. Look at the next example that all three elements of a tuple is unpacked into three variables.

```
point = (3, 4, 5)
x, y, z = point
print x + y + z
```

Sometimes you can omit parenthesis:

```
point = 3, 4, 5
x, y, z = point
print x + y + z
```

If you are asking yourself, what is the real application of unpack, I should say trust me, we need this in many different applications and I will show you in a later chapter.

3.2.6 Convert To List

You can convert a tuple to a list and vice versa.

```
point = (1, 2, 3)
#Convert tuple to list
point_list = list(point)
print type(point)
print type(point_list)

data = [4, 5, 6]
#Convert a list to tuple
data_tuple = tuple(data)
print type(data_tuple)
```

3.2.7 List of Tuples

One of the interested things about tuple is unpacking and we can use this ability within an iteration. Assume we have a list of tuples about some student information. And we want to traverse within that array and print out the result. Let's see how we can do this in a bad way and then a good pythonic way!

```
db = [
    ('tom', 23, 'math'),
    ('jack', 23, 'math'),
    ('john', 26, 'computer')
    ]

for student in db:
    print 'Name:',student[0], 'Age:', student[1], 'Course:', student[2]
```

In this type of traversing we access to each of array element one by one, including the retrieve name, age, and course by referring to their index! But this is not pythonic, let's do it again!

```
db = [
    ('tom', 23, 'math') ,
    ('jack', 23, 'math') ,
    ('john', 26, 'computer')
    ]
for name, age, course in db:
    print 'Name:',name, 'Age:', age, 'Course:', course
```

3.2.8 Benefits

Everything has pros, and cons! Tuples are significantly faster than lists in

- Construction
- Access to Elements

Immutability normally brings these advantages, but you will lose the benefit of being mutable.

Therefore, it depends on your problem, if you don't care about mutability then don't use a list! Go for tuples!

3.3 Dictionary

Dictionaries are a collection of items in the form of a (key, value) pair, meaning that each item is identified by a key, so somehow it is a **hash table** that for each item a unique key is assigned.

Lists are ordered sets of objects, whereas dictionaries are unordered sets. But the main difference is that items in a dictionariy are accessed via keys and not via their position.

Example:

```
student_record_list = ['Tom', 23, 'BSc', 'Italy', 'Java']
```

What can you tell about these five values? For instance, if Tom's language of interest is Java, or Java is the language he doesn't like? Or If you want to have his country you must always remember that its position is 3 (student_record_list[3])!! What if there are more fields! But now look at this

```
student_record_dict = {
        'name':'Tom',
        'age':'23',
        'academic_level':'BSc',
        'country':'Italy',
        'programming_language':'Java'
}
```

Now it is more clear and intelligible! If you need to know his country, just ask for `student_record_dict['country']`. And this is the purpose and benefit of dictionaries!

3.3.1 Declaration

There are several approaches to create a dictionary. Here are some of them:

Empty Dictionary

```
country_codes = {}
#or
country_codes = dict()
print country_codes
print type(country_codes)
```

Direct Initialization

```
country_codes = {'Malaysia':'MS', 'China':'ZH', 'Iran': 'IR',
'America':'US'}
print country_codes
```

A List of Binary Tuples

```
country_codes = dict([
    ('Malaysia', 'MS'),
    ('China', 'ZH'),
    ('Iran', 'IR'),
    ('America', 'US')])
print country_codes
```

3.3.2 Element Access

As mentioned earlier, access to a dictionary elements is not done by referencing an index number. We can use assigned keys to retrieve a value from a dictionary.

```
country_codes = {'Malaysia':'MS', 'China':'ZH', 'Iran': 'IR',
'America':'US'}
print country_codes['Malaysia']
```

Watch out for keys that do not exist!

In this case you will face with a **KeyError** exception!

```
country_codes = {'Malaysia':'MS', 'China':'ZH', 'Iran': 'IR',
'America':'US'}
print country_codes['Germany']
```

How Can I check if a key exists or not?

Exactly like lists or any other type od sequence.

```
country_codes = {'Malaysia':'MS', 'China':'ZH', 'Iran': 'IR',
'America':'US'}
print 'Malaysia' in country_codes
```

A better way to retrieve a value is by using `.get()`

You can read a dictionary value by using `.get(key, default)` function, that first checks for the keys existence, if it exists, it will be returned, otherwise return the default value!

```
country_codes = {'Malaysia':'MS', 'China':'ZH', 'Iran': 'IR',
'America':'US'}
print country_codes.get('Germany', 'No Data')
```

3.3.3 Keys and Values!

You can decompose a dictionary to a list of keys and values!

Keys : `.keys()`

```
country_codes = {'Malaysia':'MS', 'China':'ZH', 'Iran': 'IR',
'America':'US'}
print country_codes.keys()
```

Values : `.values()`

```
country_codes = {'Malaysia':'MS', 'China':'ZH', 'Iran': 'IR',
'America':'US'}
print country_codes.values()
```

List of binary tuple (key, value) pairs

You can also create a list of binary tuples with: `.items()`

```
country_codes = {'Malaysia':'MS', 'China':'ZH', 'Iran': 'IR',
'America':'US'}
print country_codes.items()
```

3.3.4 Traversing Dictionaries

Similar to list traversal you can traverse through a dictionary by using a `for` loop. `.items()` returns a list of binary tuples containing dictionary keys-value pairs.

Let's try:

```
country_codes = {'Malaysia':'MS', 'China':'ZH', 'Iran': 'IR',
'America':'US'}
```

```
for name, code in country_codes.items():
    print name, '=>', code
```

3.3.5 Tuples as keys!

A dictionary values can be any data type. But **keys must be immutable**!

For example, you can't use a list as a key. But what if we need to do this?! For example we want to use 2D coordinates of several points on a map as a dictionary key and then some data for that point!

Thanks to tuples, we can use a tuple instead of lists.

Now imagine we have a GEO database, let’s see what will be happen:

```
geo_db = {(3.123345, 101.1233321) : "Brown City",
          (2.123345, 103.1233321) : "Somewhere in heart of ocen!",
          (6.666666, 666.666) : "Bermuda Triangle" }
print geo_db.keys()
```

Let's try to use a list as a key!!

3.3.6 Nested Dictionary

Dictionary is the best approach to describe a real world object into computer. For example if you are talking about a book, may be you use following dictionary to describe that book.

```
book = {
    'title':'Learn Python',
    'year':2014,
    'author':'tom',
    'price':100.00
}
```

Now we have a book dictionary contains relevant information for that book!

What if this book has a list of authors?

```
book = {
    'title':'Learn Python',
    'year':2014,
    'author':['tom', 'jack', 'john'],
    'price':100.00
}
```

Actually, a dictionary can hold anything like a list! Another thing is, when dictionary may hold anything else, so a dictionary may holds another dictionary to made more complex structure.

```
book = {
     'title':'Learn Python',
     'year':2014,
     'author':['tom', 'jack', 'john'],
     'address':{
          'city':'KL',
          'street':'JS23/34.5',
          'postal':55100
     },
     'price':100.00
}
```

How we can have access to elements?

Just follow what you have learned by now. Remember you can have access to each element by its key. Now if the value is a dictionary by itself, you can continue to have access to nested element with new their keys as well.

Consider following access.

```
book = {
     'title':'Learn Python',
     'year':2014,
     'author':['tom', 'jack', 'john'],
     'address':{
          'city':'KL',
          'street':'JS23/34.5',
          'postcode':55100
     },
     'price':100.00
}

#simple access
print book['title']

#access to list
print book['author']
print book['author'][2]

#access to nested list
print book['address']['city']
print book['address']['postcode']
```

As you we can have access to city through the `print book['address']['city']` key.

Now this is possible to have a list of this kind of book dictionary, and then that would be a kind no-relational database. Some books they have address, some don't. Some books they have some new unpredicted fields! But we won't have any concern. We will talk about no-relational or no-SQL database later.

3.3.7 Modifying Dictionaries

Remove an Element

You can remove a dictionary value by referring to its key. Just use the `del` operator with a relevant key, then that element will be removed.

```
country_codes = {'Malaysia':'MS', 'China':'ZH', 'Iran': 'IR',
'America':'US'}
print country_codes['Malaysia']
del country_codes['Malaysia']
print country_codes.get('Malaysia', 'Not Found')
```

Merge two Dictionaries

You can merge two dictionaries by calling the `.update()` function. New keys will be added to the current dictionary and for those keys that already exist, their values will be updated.

```
# Sadly once more Trinket fails us. You must execute this code in your
own IDE/command line
order = {
    'id':'A123',
    'item':'C#',
    'item_type':'book',
    'qty':1,
    'state':'in_process'
}
print order

order.update({
    'state':'purchased',
    'price':200.00
})

print order
```

3.4 Set

Sets are like what you have seen in high school: Set Theory! Sets are a collection of items, with no duplication and no order between elements! And in Python sets are a mutable univocal (without repetitions) unordered collection!

```
s = set([3, 4, 4, 5, 6])
print s
```

3.4.1 Declaration

There are different approaches to create sets...

Empty Set

```
s = set()
print s
print len(s)
print type(s)
```

Direct Initialization

```
s = set([1, 2, 3, 4, 5, 4, 4, 6,7])
print s
```

Create from a List or Tuple

```
data = [1, 2, 3, 3, 'Hi']
t = (1, 2, 3, 3, 'Hi')
print set(data)
print set(t)
```

3.4.2 Element Access

When we talk about sets there is no indexing! This means you can't have access to set elements by their position. There is no order in a math set!

```
s = set([1,2,3,4,5])
print s[3]
```

However, there is a trick! First convert your set to list on the fly, then try to access to that position!

```
s = set([1,2,3,4,5])
print list(s)[3]
```

3.4.3 Append a new Element

To add a new element to a set, you can use one of two approaches. One of them is the `.add()` function of the set.

```
s = set([1,2,3,4])
s.add(5)
print s
```

And the other one is using **union** which is exactly same thing you have seen in math. But let's talk about that later.

3.4.4 Set Operations

As we said earlier, in Python set provides you the set theory! It means we have union, intersection, and other famous set theory function. Let's have a try.

Example: Imagine we have three sets of data from dozens of people! Those have been registered for INTRO workshop, WEB workshop, and DB workshop. Now we are going to have some interested reports.

```
intro = set(["Tom", "Jake", "John", "Eric"])
web = set(["Tom", "Jake", "Jill", "Mac"])
db = set(["William", "Andy"])

#Union of two sets: Find all people have registered for a workshop
print 'intro | web = ', intro.union(web)
print 'intro | web = ', intro | web

#Intersection of two sets: People have registered for both
print 'intro & web = ', intro.intersection(web)
print 'intro & web = ', intro & web

#Intersection of two sets: People have registered for both intro and db
print 'intro & db = ', intro.intersection(db)
print 'intro & db = ', intro & db

#Difference of two sets: People have registered for Intro but not for
Web
print 'intro - web = ', intro.difference(web)
print 'intro - web = ', intro - web

#Symmetric Difference between two sets: People that attend only to one
of the workshops
print 'intro ^ web = ', intro.symmetric_difference(web)
print 'intro ^ web = ', intro ^ web

#Check for dijoint or superset
if intro.isdisjoint(db):
	print "No one registered for both of intro and db"
if db.issubset(intro):
	print "All DB registrants already regoster for Intro"
if intro.issuperset(db):
	print "All DB registrants already regoster for Intro"
```

3.4.5 Modification of a set

Remove an Element

```
s1 = set([1,2,3,4,5,6])
print s1
s1.remove(6)
print s1
```

If the given element does not exist you will see an error. But there is another method **.discard()** that removes only in the case of existence, and no error will be raised.

```
s1 = set([1,2,3,4,5,6])
print s1
s1.discard(6)
print s1
s1.discard(6)
print s1
```

Remember that a **set** is mutable, that's why when you call **.discard()**, it will change its own object!

Clear a Set

```
s1 = set([1,2,3,4,5,6])
print s1
s1.clear()
print s1
```

Make a Clone

```
s1 = set([1,2,3,4,5,6])
s1_clone = s1.copy()
s1_clone.clear()
print s1
```

4 Program Design in Python

4.1 Decision Structure

4.1.1 If-Else Structure

By default, Python starts to interpret and execute your program line by line in a sequential manner! Line 1, then 2, 3 and ... But you can change this flow by creating a branch based on a condition! This is possible through the `if-else` structure. It is very common for a program that certain sets of instructions are executed conditionally, such as validating data entries, for example.

Syntax:

```
if <condition>:
    <block of code>
elif <condition>:
    <block of code>
elif <condition>:
    <block of code>
else:
    <block of code>
```

In which:

- `<condition>` : sentence that can be evaluated as `True` or `False`.
- `<block of code>` : Lines of code to be executed.
- The clauses `elif` and `else` are optional. There can be several `elifs` per `if` , but only one `else` at the end.
- Parentheses are only required to avoid ambiguity.

4.1.2 INDENTATION!

Look at the above if-else structure. In C blocks are defined by the enclosing curly brackets { and }. And the level of indentation (whitespace) before the code statements does not matter (completely optional).

In Python, the region of a code block is defined by indentation (usually a tab or four white spaces). This means that we have to be careful to indent our code correctly, or else we will get syntax errors.

```
statement1 = statement2 = True

if statement1:
    if statement2:
        print("both statement1 and statement2 are True")
```

A bad example of indentation

```
# Bad indentation!
if statement1:
    if statement2:
    print("both statement1 and statement2 are True")   # this line is not
properly indented
```

Don't Mistake. In following example both print belongs ti the same block

```
statement1 = False

if statement1:
    print("printed if statement1 is True")

    print("still inside the if block")
```

If you want the second print statement out of the block:

```
if statement1:
    print("printed if statement1 is True")

print("now outside the if block") #Remove/reduce the indentation
```

Ok, let's get back to our `if-else` with some examples:

```
weight = 85.0 #write your own weight, we are going to see how in shape
you are, don't be shy ;)
height = 1.83
bmi = weight / (height) ** 2 # your body mass index formula
print 'Your BMI is ', bmi

if bmi < 18.5:
    print 'Underweight'
elif 18.5 <= bmi <= 25:
    print 'Normal'
elif 25 <= bmi <= 30:
    print 'Overwight'
elif 30 <= bmi <= 35:
    print 'Moderatly Obese'
else:
    print 'Severely Obese'
```

There is an important point here:

Chain conditions are possible in Python. Instead of `bmi >= 25 and bmi <= 30` you can just simple write `25 <= bmi <= 30`, its cool!

If the code block is composed of only one line, it can be written after the colon:

```
if 18.5 <= bmi <= 25: print 'Normal'
```

4.1.3 Inline IF

Starting with version 2.5, Python supports the expression:

```
<variable> = <value 1> if <condition> else <value 2>
```

In which `<variable>` receives `<value 1>` if `<condition>` is true and `<value 2>` otherwise. Let's have an example. We are going to get a number from user and then print EVEN or ODD based on the given number.

Let's write a stupid program! Get a number from the user, then check if it is odd or even, and print a message!

How Can I Get a Value from User?

By now we have assigned variables ourselves, but in a more realistic model we have to read a value from user and that can be done by using `input(message)` function. Input first prints the `message` and waits for the user to enter a value and hit the enter button then that value will be returned. Exactly like `cin` in C or `Console.ReadLine()` in C#.

```
num = input("Please enter a number?")
print "Your Number is:", num
print num % 2 == 0
print "EVEN" if num % 2 == 0 else "ODD"
```

Look at the last line. `If num % 2 == 0`then we print "Even", otherwise (`else`) 'Odd'.

4.1.4 Repetition and Loops

Counter Loop with Range

Loops are repeating structures, they are generally used to parse data collections, such as lines of a file or records from a database.

One type of the loop is the **counter loop**, which is a simple type of loop that will repeat a fixed number of times. This loop called a `for` loop.

A counter `for` loop in the C language looks like this...

```
for(int i = 0; i <10; i++){
        //Your Code...
        }
```

The Pythonic version for this `for` loop can be achieved by using the `range()` function. Remember that `range(start, stop, step)` generates a sequence of integer numbers, and you can use it as counter for your loop.

```
for i in range(10):
    print 'i :', i
print "-----------"
for i in range(10, 20, 2):
    print 'i :', i
```

4.1.5 Traverse Collections

In this version, we have a sequence or collection of items, and we are going to iterate through the sequence and play with all of the items one by one. Let's see an example...

```
text = "The use of COBOL cripples the mind; its teaching should,
therefore, be regarded as a criminal offense."
words = [] # [w.strip(' .') for w in text.split(' ')]
for word in text.split(' '):
    words.append(word.strip(' .').lower())
print words
```

So look at this `for word in text.split(' '):`. First, `text.split(' ')` returns back a list of words that are generated by splitting the text based on whitespace. Then our loop starts to iterate through this list and pick up those words one by one, and return it back for you within the `word`.

If you are familiar with `foreach` in C# or Java, may be you find it similar.

Actually whenever you don't care about the position of an element inside of a collection, there's no need to use a counter loop:

```
text = "The use of COBOL cripples the mind; its teaching should,
therefore, be regarded as a criminal offense."
tokens = text.split(' ')
words = []
for i in range(len(tokens)):
    words.append(tokens[i].strip(' .').lower())
print words
```

But this doesn't have any point!

Let me add something else, remember **List Comprehension**, and how beautiful they are. Most of the time whenever you don't want to do many things inside of your loop, you can use list comprehension. Let's rewrite our example by using list comprehension.

```
text = "The use of COBOL cripples the mind; its teaching should,
therefore, be regarded as a criminal offense."
words = [w.strip(' .').lower() for w in text.split(' ')]
print words
```

Iterate a range

```
for i in range(-3, 3):
    print i
```

Iterate a collection:

```
for w in ["Python", "is", "awesome"]:
    print w
```

Iterate a collection with indexing

```
for idx, w in enumerate(["Python", "is", "awesome"]):
    print idx, ': ', w
```

Iterate a tuple

```
for x in (1, 2, 3, 4, 'Hi', 4.5):
    print x
```

Iterate a Dictionary

```
country_codes = {'Malaysia':'MS', 'China':'ZH', 'Iran': 'IR',
'America':'US'}
for key, value in country_codes.items():
    print key,' : ', value
```

4.1.6 Conditional Loop

`while` simply executes a block of code until a given condition is met. The `while` loop is appropriate when there is no way to determine how many iterations will or need to occur.

Syntax:

```
while <condition>:
    <block of code>
```

```
        Carry
        break
    else:
        <block of code>
```

The block of code inside the `while` loop is repeated as long as the loop condition is being evaluated as true.

Example:

```
n = input("Enter the upper boundry of your fibonucci sequence? ")
a = 0
b = 1
print a
while b < n:
    print b
    t = b
    b = a + b
    a = t
```

You can make this code more Pythonic, think about it!

4.1.7 Break and Continue

You can control the iteration of a `for` loop. You can stop it or jump to the next cycle! This can be done with the `break` and `continue` commands.

Let's see an example.

```
for i in range(10):
    if i % 2 == 0:
        continue
    elif i == 8:
        break
    else:
        print i
```

As you see whenever we have even numbers, we jump to the next cycle, and when we reach 8, we meet `break` which stops the whole lop.

4.2 Functions

We use functions to have modularity. It means we put our valuable code into these boxes to reuse them again and again! In this way we will bring a sort of logical building block to our application.

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusability.

As you already know, 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.

4.2.1 Declare It!

A function in Python is defined using the keyword `def`, followed by a function `name`, a signature within parentheses `()`, and a colon `:`. The following code, with one additional level of indentation, is the function body.

Let's see an example...

```
def circle(radius):
          area = radius ** 2 * 3.14
          return area

print circle(10)
a = circle(5)
print 'Area', a
```

4.2.2 Call with Keyword Argument

Another way to call functions is with keyword arguments

```
def max(x, y):
    m = x if x >=y else y
    return m

print max(10, 5)
print max(y=10, x=5)
```

Keyword arguments can arrive in any order.

You will see the main benefit of keyword arguments in next section.

4.2.3 Doc String

Doc Strings are strings that are attached to function in Python. In functions, Doc Strings are placed inside the function body, usually at the beginning.

The objective of Doc Strings is to serve as documentation for that structure. Doc strings are useful in different ways. Whenever you want to remember how the function does its job and what input parameters it takes use the `help(function name)` call.

```
def circle(radius):
    """
    Calculates circle area for the given radius
```

```
        """
        area = 3.14 * radius ** 2
        return area
    help(circle)
```

4.2.4 Default Arguments

In definition of a function, we can give default values to the arguments the function takes:

```
    def circle(radius, pi = 3.14):
        """
        Calculates circle area for the given radius
        """
        area = pi * radius ** 2

        return area
```

Now you can call **circle** without pass anything for the **pi** argument then the default value will be selected, or you can pass a new value and override the default version.

```
    def circle(radius, pi = 3.14):
        """
        Calculates circle area for the given radius
        """
        area = pi * radius ** 2
        return area

    print circle(10)
    print circle(10, 2.6)

    #If you try to call argument with keyword style, then order does not
    matter
    print circle(pi = 2.6, radius = 10)
```

This is a big mistake* circle(pi = 2.5, 10)*, try to avoid this.

4.2.5 Multiple Parameters

You can define functions that take a variable number of positional arguments.

```
    def average(*args):
        print 'Arguments:',args
        return sum(args)/float(len(args))

    print average(3, 4, 5)
    print average(3, 4, 5, 2, 3)
```

You can define functions that take a variable number of keyword arguments, as well

```
def create_book(**kwargs):
    print 'Arguments:',kwargs
    #Connect databse as add specified book
    print 'New book is saved'


create_book(isdn="123", title="C#", author="tom")
create_book(isdn="124", title="Java", author="tom", year=2005)
```

4.2.6 Multiple Output

In Python a function is possible to returns more than one value.

We have two functions, the first calculates area and the second one circumference.

```
def circle_area(r):
    return r **2 * 3.14

def circle_circumference(r):
    return r * 2 * 3.14
```

Now let's wrap both operations into one function.

```
def circle(r):
    area = r ** 2 * 3.14
    cir = r * 2 * 3.14
    return area, cir

a, c = circle(10)
print 'Area:', a, ', Circumference:', c
```

4.2.7 Anonymous Function

In Python we can also create unnamed or anonymous functions, using the lambda keyword:

```
f1 = lambda x: x**2

# is equivalent to

def f2(x):
    return x**2

print f1(5)
print f2(5)
```

Anonymous functions (first class) are building block for functional programming, and we will cover them in next chapter.

Let's have another example.

```
add = lambda x, y: x + y
print add(5, 6)
```

Ok, and let's have another dirty example. You can create an anonymous function and execute it without assigning it to a variable.

```
print (lambda x, y: x + y)(4, 5)
```

4.3 Functional Programming

4.3.1 High Order Function

In computer science, functional programming is a programming paradigm, a style of building the structure and elements of computer programs that treats computation as the evaluation of mathematical functions and avoids **state and mutable data**.

Functional programming emphasizes functions that produce results that depend only on their inputs and not on the **program state**—i.e. **pure mathematical functions**.

It is a **declarative programming paradigm**, which means programming is done with expressions.

In functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function `f` twice with the same value for an argument `x` will produce the same result `f(x)` both times.

Eliminating **side effects**, i.e. changes in state that do not depend on the function inputs, can make it much easier to understand and predict the behaviour of a program, which is one of the key motivations for the development of functional programming.

One of the main important things in functional programming is high-order functions. A high order function is a function that may accepts another function as an input or returns a function as output!

Let's see an example. We want to calculate power 2 of all numbers from 0 until 10. We can do this by using `for` loops, but in functional programming style we use `map` function:

```
print map(lambda x: x ** 2, range(0, 10))
```

Another example, from a range of numbers we want to keep only even numbers, again we can do this with `for` loop and `if`, but in functional programming we have `filter`:

```
print filter(lambda x: x % 2 == 0, range(1, 10))
```

We can redo both examples, with list comprehension as well

```
print [x ** 2 for x in range(1, 10)]

print [x for x in range(1, 10) if x % 2 == 0]
```

4.4 Modules

Python contains many built-in functions that are provided as language libraries. When you install Python, it comes with many different libraries for different tasks. Also, you can add third-party libraries (from PyPi) and increase the number of libraries you have access to while coding.

Before that, you have to import them into your code. In C++ `#include`, in C# `using`, in Java and Python `import`.

For example we want to import the `math` library and use some of its functions.

```
import math
print math.pi
print math.sqrt(3)
```

Instead of bringing all math libraries to your code and make your global scope messy, you can import selected functions:

```
from math import ceil, floor
print ceil(3.7)
print floor(3.7)
```

If you are lazy and its difficult for you to write `math` again and again, you can choose a shorter alias.

```
import math as m
print m.pi
```

Python modules are just ordinary Python files. You can write your own, and import them. The name of the module is the same as the name of the file.

How Can I find out what functions are wrapped inside of a module?

In python whenever you want to make a list of available functions associated with an object, class, or module use `dir(objet or module name)`.

```
import math
dir(math)
```

How Can I have list of all available modules in Python?

```
import sys
sys.builtin_module_names
```

How Can I have a help comment for a function or module?

Simple try: `help(function, object or module name)`

4.4.1 Create Your Own Module

One of the most important concepts in good programming is to reuse code and avoid repetition. The idea is to write functions and classes with a well-defined purpose and scope, and reuse them instead of repeating similar code in different parts of a program (modular programming). The result is usually that readability and maintainability of a program is greatly improved. What this means in practice is that our programs have fewer bugs, are easier to extend, debug, and troubleshoot.

Python supports modular programming at different levels. Functions and classes are examples of tools for low-level modular programming. Python modules are a higher-level modular programming construct, where we can collect related variables, functions and classes in a module. A python module is defined in a python file (with file-ending .py), and it can be made accessible to other Python modules and programs using the import statement.

Consider the following example: the file `mymodule.py` contains simple example implementations of a variable, function, and a class:

```
"""
Example of a python module. Contains a variable called my_variable,
a function called my_function, and a class called MyClass.
"""

my_variable = 0

def my_function():
    """
    Example function
    """
    return my_variable

class MyClass:
    """
    Example class.
    """

    def __init__(self):
        self.variable = my_variable
    def set_variable(self, new_value):
        """
        Set self.variable to a new value
        """
        self.variable = new_value

    def get_variable(self):
        return self.variable
```

Now save this file into **mymodule.py**, then create another python file in the same directory and at the first line you can import your module.

```
import mymodule
```

Even you can use **help()** function to get Doc String of your module.

```
help(mymodule)
```

Remember that in this module you have one global variable **my_variable** and one function **my_function** and one class **my_class**. So you can use any of them like below.

```
print mymodule.my_variable
print mymodule.my_function()

my_class = mymodule.MyClass()
my_class.set_variable(10)
my_class.get_variable()
```

You can make some dynamic changes to your module. For example you can execute **mymodule.my_variable = 20**, then if you want to reset to its default value, you can use **reload(mymodule)**.

4.5 Object Oriented Programming

4.5.1 Class

Creating Classes

Classes are the key feature of object-oriented programming. A class is a structure for representing an object and the operations that can be performed on the object. In Python a class can contain attributes (variables) and methods (functions). A class is defined almost like a function, but using the **class** keyword, and the class definition usually contains a number of class method definitions (a function in a class).

Each class method should have an argument **self** as it first argument. This object is a self-reference.

Some class method names have special meanings, for example:

- **__init__**: The name of the method that is invoked when the object is first time created.
- **__str__** : A method that is invoked when a simple string representation of the class is needed, as for example when printed.
- You can find many more here

Let's create our first class to show a **2D Point** with two properties of X, and Y.

```
class Point:
    """
    A simple class for representing a point in a Cartesian coordinate
system.
    """

    def __init__(self, x, y):
        """
        Create a new Point at x, y.
        """
        self.x = x
        self.y = y

    def translate(self, dx, dy):
        """
        Translate the point by dx and dy in the x and y direction.
        """
        self.x += dx
        self.y += dy

    def __str__(self):
        return("Point at [%f, %f]" % (self.x, self.y))

# this will invoke the __init__ method in the Point class
p1 = Point(0, 0)

# this will invoke the __str__ method
print p1

# to access class method
p2 = Point(1, 1)

p1.translate(0.25, 1.5)

print(p1)
print(p2)
```

Note that calling class methods can modify the state of that particular class instance, but does not affect other class instances or any global variables.

That is one of the nice things about object-oriented design: code such as functions and related variables are grouped in separate and independent entities.

4.5.2 Inheritance

In Python, binding an inheritance relationship between classes has two steps. First, you have to mention the parent class name, then inside of a child constructor, you have to call the parent constructor.

```
class Point:
    """
    Simple class for representing a point in a Cartesian coordinate
system.
    """

    def __init__(self, x, y):
        """
        Create a new Point at x, y.
        """
        self.x = x
        self.y = y

    def translate(self, dx, dy):
        """
        Translate the point by dx and dy in the x and y direction.
        """
        self.x += dx
        self.y += dy

    def __str__(self):
        return("Point at [%f, %f]" % (self.x, self.y))

    def __repr__(self):
        return("Point at [%f, %f]" % (self.x, self.y))


class Point3D(Point):
    def __init__(self, x, y, z):
        Point.__init__(self, x, y)
        self.z = z

    def translate(self, dx, dy, dz):
        Point.translate(self, dx, dz)
        self.z += dz
    def __str__(self):
        return("Point at [%f, %f, %f]" % (self.x, self.y, self.z))

    def __repr__(self):
        return("Point at [%f, %f, %f]" % (self.x, self.y, self.z))

p = Point3D(4, 5, 2)
print p

p.translate(1, 1, 1)
print p
```

5 Advanced Topics

5.1 File Input/Output

5.1.1 Create a File

Trust me! One of the simplest things in Python is creating a file! First step is creating a file handler and then start to write your data, finally you have to close your file handler.

Open a File Handler with `open(path, mode)`

```
f = open(r'c:\pycademy\memo.txt', 'w')
```

First argument is your file name and its path, then second parameter is your file access mode. We have the following modes to access open files.

- `r` only READ access
- `rb` only READ access from a binary files
- `w` only WRITE access
- `wb` only WRITE access to a binary file
- `w+` READ, and WRITE access to a file. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
- `a` only APPEND to a file
- `ab` only APPEND to a file which is binary
- `a+` both APPEND and READ

Now Write into the File using `write()`

```
f.write("Providing better computer science education in public schools
to kids, and encouraging girls to participate, is the only way to
rewrite stereotypes about tech and really break open the old-boys'
club.")
```

And finally you have to close your file.

```
f.close()
```

5.1.2 Read a File

Similar to **.write**, you can easily read from a file. Again you have to open a file with read access, and then call the **read** method.

To read a file you have different approaches:

Read Entire File

```
f = open(r'c:\pycademy\memo.txt', 'r')
print f.read()
f.close()
```

What if the file you are reading is too large to be loaded into the memory?

Read specified number of bytes

```
f = open(r'c:\pycademy\memo.txt', 'r')
print f.read(10)
f.close()
```

In this approach we can start by reading X bytes of the file, then process that amount, and then move on to the next X bytes. Keep doing this until you reach to the end of file, then the return value would be an empty string! Hence, you can easily and efficiently process large files.

Open Safe

If you are like me and forget to close the file, there is a solution to help you. Consider the following code:

```
with open(r'c:\pycademy\memo.txt', 'r') as f:
    print f.read()
print "I am happy now"
```

In this version the **with** will manage closing the file handler. So when you finish your **IO** task the handler will be closed automatically.

5.1.3 Some OS Level Methods

In this section I will talk about some relevant **OS** level functionalities that we need for any IO programming.

Check Whether a Given Path Exists or Not!

```
import os
if os.path.exists(r'c:\pycademy\memo.txt'):
    #do your task
```

Get some Statistics about a File

```
import os
if os.path.exists(r'c:\pycademy\memo.txt'):
    stat = os.stat(r'c:\pycademy\memo.txt')
    print stat
    print stat.st_size
```

Remove a File

```
import os
if os.path.exists(r'c:\pycademy\memo.txt'):
    os.remove(r'c:\pycademy\memo.txt')
```

Create and Remove Directory

```
import os
#to create
os.mkdir(r'c:\test')
#to remove
os.rmdir(r'c:\test')
```

Get List of Files and Directories

```
import os
items = os.listdir(r'c:/')
print items
```

Search for File based on a Pattern

`glob` is a library that provides an easy approach to get lists of files inside of a directory based on a given filter criteria.

The following code returns back all of the files with the extension `.jpg` within in the `user` directory.

```
import glob
files = glob.glob('c:/user/*.jpg')
print files
```

`glob` returns a list of strings that each item contains the file name and its complete pat.

5.2 Exceptions Handling

In Python errors are managed with a special language construct called "Exceptions". When errors occur exceptions can be raised, which interrupts the normal program flow and falls back to somewhere else in the code where the closest `try-except` statement is defined.

To gracefully catch errors that are generated by functions and class methods, or by the Python interpreter itself, use the `try` and `except` statements:

```
try:
    # normal code goes here
except:
    # code for error handling goes here
    # this code is not executed unless the code
    # above generates an error
```

Remember an error is raised when you try to access to a dictionary with a key that does not exist! You can manage this with `try-exception` block:

```
country_codes = {'Malaysia':'MS', 'China':'ZH',
'Iran': 'IR', 'America':'US'}
try:
    print country_codes['Italy']
except:
    print "Key doesn't exist"
```

You can have more information about the raised exception. See the following examples:

```
try:
    5 / 0
except Exception as e:
    print "Error:", e
    pass
```

In this example **e** holds some information regarding the raised exception.

5.3 Time

This module provides a number of functions to deal with dates and the time within a day. It’s a thin layer on top of the C runtime library.

A given date and time can either be represented as a floating point value (the number of seconds since a reference date, usually January 1st, 1970), or as a time tuple.

5.3.1 Current Time

```
import time
now = time.time()
print now
```

```
=>
1404651432.99 seconds since (1970, 1, 1, 0, 0, 0)
```

There are many other things you can do with `time` module. For more information check here

5.3.2 DateTime

DateTime is a module that simply provides date and time :) You can have access to current date and time, or you can create your own date and time object. You can use these object in your application to compare different date, database query and other purposes. Objects of the **datetime** type represent a date and a time in some time zone.

The module contains the following types:

- The **datetime** type represents a date and a time during that day.
- The **date** type represents just a date, between year 1 and 9999 (see below for more about the calendar used by the **datetime** module)
- The **time** type represents a time, independent of the date.
- The **timedelta** type represents the difference between two time or date objects.
- The **tzinfo** type is used to implement **timezone** support for time and **datetime** objects; more about that below.

datetime

```
import datetime
date = datetime.datetime(2014, 7, 6, 20, 50, 10)
print date
print date.year, date.month, date.day
print date.hour, date.minute, date.second
print date.microsecond
```

You can also create **datetime** objects by using one of the many built-in factory functions (but note that all such functions are provided as class methods, not module functions):

```
import datetime
import time
```

```
print datetime.datetime(2014, 7, 6, 21, 41, 43)

print datetime.datetime.today()
print datetime.datetime.now()
print datetime.datetime.fromtimestamp(time.time())
```

```
=>
2014-07-06 21:41:43
2014-07-06 20:53:33.465000
2014-07-06 20:53:46.981000
2014-07-06 20:53:59.268000
```

The **datetime** type provides other formatting methods as well, including the highly general **strftime** method

```
import datetime
import time

now = datetime.datetime.now()

print now
print now.ctime()
print now.isoformat()
print now.strftime("%Y%m%dT%H%M%S")
```

```
=>
2014-07-06 20:54:12.586000
Sun Jul 6 20:54:12 2014
2014-07-06T20:54:12.586000
20140706T205412
```

5.4 Regular Expression

A regular expression is a special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern. Regular expressions are widely used in UNIX world.

Python provide a great library called `re` to provide regular expression functionality.

In general we need following abilities:

- Find and match pattern within a large text based on a regular expression
- Search for a pattern based on regular expression
- Split a text by using a regular expression as delimiter
- Replace patterns defined by a regular expression with a replacement

- Compile a regular expression into a Regex Object

Match

This function attempts to match RE pattern to string with optional flags.

Here is the syntax for this function:

```
re.match(pattern, string, flags=0)
```

Here is the description of the parameters:

- **pattern:** This is the regular expression to be matched.
- **string:** This is the string, which would be searched to match the pattern at the beginning of string.
- **flags:** You can specify different flags using bitwise OR (|). These are modifiers, which are listed in the table below:

Modifier	Description
re.I	Performs case-insensitive matching.
re.L	Interprets words according to the current locale. This interpretation affects the alphabetic group (\w and \W), as well as word boundary behavior (\b and \B).
re.M	Makes $ match the end of a line (not just the end of the string) and makes ^ match the start of any line (not just the start of the string).
re.S	Makes a period (dot) match any character, including a newline.
re.U	Interprets letters according to the Unicode character set. This flag affects the behavior of \w, \W, \b, \B.
re.X	Permits "cuter" regular expression syntax. It ignores whitespace (except inside a set [] or when escaped by a backslash) and treats unescaped # as a comment marker.

The `re.match` function returns a match object on success, None on failure. We would use `group(num)` or `groups()` function of match object to get matched expression.

```
import re
line = "Cats are smarter than dogs"
matchObj = re.match( r'(.*) are (.*?) .*', line, re.M|re.I)
if matchObj:
   print "matchObj.group() : ", matchObj.group()
   print "matchObj.group(1) : ", matchObj.group(1)
   print "matchObj.group(2) : ", matchObj.group(2)
else:
   print "No match!!"
```

When the above code is executed, it produces following result:

```
matchObj.group() :   Cats are smarter than dogs
matchObj.group(1) :   Cats
matchObj.group(2) :   smarter
```

Search

This function searches for first occurrence of RE pattern within string with optional flags.

Everything about this method is exactly same as `match` except one different. Python offers two different primitive operations based on regular expressions: match checks for a match only at the beginning of the string, while search checks for a match anywhere in the string (this is what Perl does by default).

So to check or search for a pattern do NOT use `match` as it doesn't check anywhere.

Replace

Some of the most important re methods that use regular expressions is sub.

SYNTAX:

```
re.sub(pattern, repl, string, max=0)
```

This method replaces all occurrences of the RE pattern in string with `repl`, substituting all occurrences unless max provided. This method would return modified string.

EXAMPLE: Following is the example:

```
import re

phone = "2004-959-559 # This is Phone Number"

# Delete Python-style comments
num = re.sub(r'#.*$', "", phone)
print "Phone Num : ", num

# Remove anything other than digits
num = re.sub(r'\D', "", phone)
print "Phone Num : ", num
```

When the above code is executed, it produces the following result:

```
Phone Num :  2004-959-559
Phone Num :  2004959559
```

Compile

Whenever you need to use any of the above functions frequently for the same patter, to save time and improve the performance, it's better to create a REGEX Object.

Actually every time you call any of the above functions, first Python parse the given pattern and compile it into an intermediate object, then use that one for pattern matching. So if you are inside of a for loop that repeats million times, and you are searching for a same pattern among million different text, then each cycle you will waste a significant amount of time for each dummy recreation of a REGEX Object.

In these cases the rational way is to create one REGEX object before for loop and then use it again and again! It's a kind of caching!

Anyway look at the following example:

```
import re

comment_matcher = re.compile(r'#.*$')
phone = "2004-959-559 # This is Phone Number"
print re.sub(comment_matcher, "", phone)

phone = "2034-959-559 # This is Phone Number"
print re.sub(comment_matcher, "", phone)
```

```
=>
2004-959-559
2034-959-559
```

As you can see we got the same result, but for the second time we didn't create another pattern again and again!

5.5 Compare Sequences

To wrap up whatever we told you regarding python sequences, please follow this link to have a generalized comparison between python sequences.

5.6 Problem Set: Time for Challenge!

Now, it is a good time to challenge what you have learned. In this section, I will share dozens of programming problems that must be implemented with Python. I am not going to give you the answers. I need you focus and try to answer all of them, if you couldn't then you may contact me through my email.

Ok let's start....

5.6.1 Arithmetic

For this section, you have to create a file called `mylib.py` which is going to be your own module. Then implement your code inside of this module. We are going to use this as your library in next section.

For each one of the following problems, design and implement an algorithm using Python.

1. Bubble Sort
2. Merge Sort
3. List of Number Divisors
4. Prime Number Checker
5. Frequency counter within a word bag

5.6.2 Python Library

1. Install `pip` and `setup_tools`
2. Install `pyquery`
3. Install `mailer`

5.6.3 Files and Folders

1. Inside the Pycademy folder, create a sub folder `pics` and then copy some images into the folder.
2. Use `ZipFile` zip `pics` folder and name it 'pic.zip'
3. Send an email and attach the zipped file

5.6.4 Html

- Download HTML for the following link
  http://en.wikipedia.org/wiki/Python_(programming_language)
- Extract Text from Html (Using PyQuery)
- Remove Stop words (Stop words are available inside of the PyCademy folder in file stopwords.txt)
- Calculate the frequency distribution for each word.