The Basics

- Python usa indexación (espacios, tabuladores...) para definir bloques de código.
- La diferencia entre python v2 y python v3 es grande. v2 está más extendida por lo general. Ejemplo: **print**.
- Python is completely object oriented, and not "statically typed". You do not need to declare variables before using them, or declare their type. Every variable in Python is an object.
- Comentarios con `#`

`print` examples

```
myint = 1
mystr = 'Hola Mundo'

print myint+1
# spaces?
print myint, mystr
print "number:", myint, "-", mystr

myPI = 3.14159265359
print 'The value of PI is approximately %5.3f.' % myPI

# Mixing operators between numbers and strings is not supported:
print myint + 2 + mystr
```

Numbers

```
myint = 5
myfloat = 5.0
# or...
myfloat1 = float(5)
```

Strings

Es válido tanto `'Hello'` como `"World"` pero...

```
mystring = "Don't worry about apostrophes"
```

Concatenación de strings

```
hello = "hello"
world = "world"
helloworld = hello + " " + world
```

o repetición...

```
lotsofhellos = "hello" * 3
```

Algunas operaciones con string

```
mystr = 'Hello World'

len(mystr)              # 11

mystr.count('o')        # 2

mystr.index('o')        # 4
mystr.index('z')        # ValueError: substring not found !!!!!

mystr [2:8]             # 'llo Wo'
mmystr[1:-1]            # 'ello Worl'
```

Hay todo tipo de operaciones que se pueden hacer con strings, algunas son: `.lower()`, `.upper()`, `.split()`, `.startswith()`, `.endswith()`...

Assignments

```
# OK:
a = 1
b, c = 2, 3
d, str = 4, 'Hola Mundo'

# ERRROR:
c, d = 5
```

List

Similares a los arrays. Pueden contener varios tipos.

```
mylist = []  # empty
mylist.append(10)
mylist.append(2.5)
mylist.append('Hello World')

for x in mylist :
    print x
```

También se permite concatenación de listas con el operador `+`

```
evennumbers = [2,4,6]
oddnumbers = [1,3,5]

allnumbers = evennumbers + oddnumbers
print allnumbers  # [2, 4, 6, 1, 3, 5]
```

Igual que con los strings, se pueden concatenar listas con el operador `*`

```
print [1,2,3] * 3

# [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

`if` statement

```
if <condition> :
    <do something>
elif <condition> :
    <do something>
else :
    <do something>
```

Boolean operators

- `True`, `False`
- `and`, `or`, `not`
- `==` y `!=`
- Otros operadores: `in`, `is`, `not`

El operador `==` comprueba la igualdad de dos variables:

```
x = [1,2,3]
print x == [1,2,3]      # True
print x == [3,2,1]      # False
```

Mientras que el operador `is` comprueba la igualdad de las instancias que se comprueban:

```
x = [1,2,3]
y = x
z = [1,2,3]
print x == y            # True
print x is y            # True

print x == [1,2,3]      # True
print x is [1,2,3]      # False !!!

print x == z            # True
print x is z            # False !!!
```

El operador `in` comprueba si un objeto existe dentro de otro objeto contenedor iterable:

```
if name in ["John", "Rick"]:
    print "Your name is either John or Rick."


if not name in ["John", "Rick"]:
    print "Your name is neither John or Rick."
```

Loops

Bucle `for`:

```
# Prints out the numbers 0,1,2,3,4
for x in range(5) :
    print x

# Prints out 3,5,7
for x in range(3, 8, 2) :
    print x
```

La función `range(3,8,2)` devuelve la lista `[3, 5, 7]`. También tenemos la función `xrange(3,8,2)` devuelve un iterador, lo cual es más eficiente para usar con un `for`.

Bucle `while`:

```
count = 0
while count < 5 :
    print count
    count += 1
```

También existen las sentencias `break` y `continue`.

Functions

```
def foo() :
    print "Hello World!"


def bar(username, greeting) :
    str = "Hello, %s, I wish you %s" % (username, greeting)
    return str

print bar('alber', 'a great keynote!')
```

Class

Definición de una clase:

```
class MyClass :
    foo = 'Hello World'

    def foox2(self) :
        self.foo *= 2

    def bar(self) :
        print self.foo
```

Creación de un objeto de dicha clase:

```
myobj = MyClass()

print myobj.foo         # 'Hello World'

myobj.foox2()

print myobj.bar()       # 'Hello WorldHello World'
```

Dictionaries

Objetos con relaciones clave-valor. Las claves pueden ser de distintos tipos.

```
mydict = {}
mydict['foo'] = 'bar '
mydict[3] = mydict['foo'] * 3

print mydict            # {3: 'bar bar bar ', 'foo': 'bar '}
```

Para iterar sobre un diccionario se puede hacer similar a una lista:

```
mydict = {
    'foo' : 'bar',
    1 : 1234,
    'mykey' : 'myvalue'
}

for key, value in mydict.iteritems() :
    print key, ':', value

# prints out:
# 1 : 1234
# foo : bar
# mykey : myvalue
```

Para eliminar un valor:

```
del mydict[1]
print mydict        # {'foo': 'bar', 'mykey': 'myvalue'}
```

Imports

Para importar un paquete se usa la palabra clave `import` + el nombre del módulo a importar.

```
import math             # import math library

print print math.pi     # 3.14159265359
```

Con la función `dir(math)` podemos ver las funciones implementadas en el módulo `math`.
Con la función `help(math)` tenemos acceso a la documentación de los módulos.

Exceptions handling

La captura de excepciones se realiza con un bloque `try-catch`. Un ejemplo trivial en el que se quiere iterar sobre una lista de 20 elementos pero la lista viene dada por el usuario, por ejemplo. Así que nosotros capturaríamos la excepción y añadiríamos tantos ceros como elementos falten:

```
def do_stuff_with_number(n):
    print n

user_list = (1, 2, 3, 4, 5)

for i in range(20):
    try:
        do_stuff_with_number(user_list[i])
    except IndexError:          # Raised when accessing a non-existing index
of a list
        do_stuff_with_number(0)
```