Python data types

In this part of the Python programming tutorial, we will talk about Python data types.

Computer programs work with data. Spreadsheets, text editors, calculators or chat clients. Tools to work with various data types are essential part of a modern computer language. According to the wikipedia definition, a *data type* is a set of values, and the allowable operations on those values.

Python has a great set of useful data types. Python's data types are built in the core of the language. They are easy to use and straightforward.

Boolean values

There is a duality built in our world. There is a Heaven and Earth, water and fire, jing and jang, man and woman, love and hatred. In Python programming language, the Boolean datatype is a primitive datatype having one of two values: True or False. This is a fundamental data type. Very common in computer programs. Interestingly, this data type was not there from the beginning, but it was created later on.

Happy parents are waiting a child to be born. They have chosen a name for both possibilities. If it is going to be a boy, they might have chosen John. If it is going to be a girl, they might have chosen Victoria.

```
#!/usr/bin/python

# kid.py

import random

male = False
male = bool(random.randint(0, 1))

if (male):
   print "We will use name John"
else:
   print "We will use name Victoria"
```

The script uses a random integer generator to simulate our case.

```
 import random
```

Here we import the random module that is used to calculate random numbers.

```
 male = bool(random.randint(0, 1))
```

Here we use two functions. the `randint()` function returns a random number from the given integer boundaries. In our case 0 or 1. The `bool()` function converts the integers to boolean values.

```
if (male):
```

```
   print "We will use name John"
else:
   print "We will use name Victoria"
```
We print the name. The if command works with boolean values. If the `male` is True, we print the "We will use name John" to the console. If it has a False value, we print the other string.

The following script shows some common values that are considered to be True or False.

```
#!/usr/bin/python

# boolean.py

print bool(True)
print bool(False)
print bool("text")
print bool("")
print bool(' ')
print bool(0)
print bool()
print bool(3)
print bool(None)
```
In fact, in the previous example, we did not have to use the `bool()` function.

```
$ ./boolean.py
True
False
True
False
True
False
False
True
False
```
This is the output of the `boolean.py` script.

None

There is another special data type - `None`. Basically, the data type means non existent, not known or empty.

```
#!/usr/bin/python

# None.py

def function():
   pass

print function()
```
In our example, we define a function. Functions will be covered later in the tutorial. The function does nothing. It does not explicitly return any value. Such a function will implicitly return None object.

```
None
```
This is the output of the `None.py` script.

Numbers

In Python programming language, we have integer numbers, floating point numbers, and complex numbers.

If we work with integers, we deal with discrete entities. We would use integers to count apples.

```
#!/usr/bin/python

# apples.py

# number of baskets
baskets = 16

# number of apples in a basket
apples_in_basket = 24

# we get the total number of apples
total = baskets * apples_in_basket


print "There are total of", total, "apples"
```
In our script, we count the total amount of apples. We use the multiplication operation.

```
$ ./apples.py
There are total of 384 apples
```
The output of the script.

Floating point numbers represent real numbers in computing. Real numbers measure continuous quantities. Let's say a sprinter for 100m ran 9.87s. What is his speed in km/h?

```
#!/usr/bin/python

# sprinter.py

# 100m is 0.1 km

distance = 0.1

# 9.87s is 9.87/60*60 h

time = 9.87 / 3600

speed = distance / time

print "The average speed of" \
      " a sprinter is " , speed, "km/h"
```
To get the speed, we divide the distance by the time.

```
print "The average speed of" \
      " a sprinter is " , speed, "km/h"
```
The `\` character is called an escape character. The escape character changes the meaning of the new line character. We see the code in two lines, but the interpreter drops the new line character, and sees only one line. Without the escape character, the interpreter would complain about indentation.

```
$ ./sprinter.py
The average speed of a sprinter is  36.4741641337 km/h
```
This is the output of the sprinter script. Value 36.4741641337 is a floating point number.

Strings

String is a data type representing textual data in computer programs. Probable the single most important data type in programming.

Strings in Python can be created using single quotes, double quotes, and triple quotes. When we use triple quotes, strings can span several lines without using the escape character.

```
#!/usr/bin/python

# strings.py

a = "proximity alert"
b = 'evacuation'
c = """
requiem
for
a
tower
"""

print a
print b
print c
```
In our example we assign three string literals to `a`, `b`, and `c` variables. And we print them to the console.

```
$ ./strings.py
proximity alert
evacuation

requiem
for
a
tower
```
This is the output of the strings.py script.

When we work with strings, we can use *escape sequences*. The escape sequences are special characters that have a specific purpose when used within a string.

```
print "   bbb\raaa" # prints aaabbb
```
The carriage return `\r` is a control character for end of line return to beginning of line.

```
#!/usr/bin/python

# strophe.py

print "Incompatible, it don't matter though\n'cos someone's bound to hear my
cry"
print "Speak out if you do\nYou're not easy to find"
```
The new line is a control character, which begins a new line of text.

```
$ ./strophe.py
Incompatible, it don't matter though
'cos someone's bound to hear my cry
Speak out if you do
You're not easy to find
```
Next we examine the backspace control character.

```
print "Python\b\b\booo" # prints Pytooo
```
The backspace control character `\b` moves the cursor one character back. In our case, we use three backspace characters to delete three letters and replace them with three o characters.

```
print "Towering\tinferno" # prints Towering        inferno
```
The horizontal tab puts a space between text.

```
"Johnie's dog"
'Johnie\'s dog'
```
Single and double quotes can be nested. Or in case we use only single quotes, we can use the backslash to escape the default meaning of a single quote.

```
print "eagle has", len("eagle"), "characters"
```
We can use the `len()` function to calculate the length of the string in characters.

If we append an `r` to the string, we get a raw string. The escape sequences are not interpreted.

```
#!/usr/bin/python

# raw.py

print r"Another world\n"
$ ./raw.py
Another world\n
```
We get the string with the new line character included.

In the next example, we will show string multiplication and concatenation.

```
#!/usr/bin/python

# strings2.py
```

```
print "eagle " * 5

print "eagle " "falcon"

print "eagle " + "and " + "falcon"
```
The `*` operator repeats the string n times. In our case five times. Two string literals next to each other are automatically concatenated. We can also use the `+` operator to explicitly concatenate the strings.

```
$ ./strings2.py
eagle eagle eagle eagle eagle
eagle falcon
eagle and falcon
```
This is the output of the `strings.py` script.

Python programming language has several built-in data types for working with collections of values. These are currently tuples, lists, sets, and dictionaries.

Tuples

A *tuple* is an immutable sequence data type. The tuple can contain mixed data types.

```
fruits = ("oranges", "apples", "bananas")
```
Tuples are created using round brackets. Here we have a tuple consisting of three fruit types.

```
fruits = "apples", "oranges", "bananas"
print fruits  # prints  ('apples', 'oranges', 'bananas')
```
The parentheses are not mandatory. We can omit them.

```
#!/usr/bin/python

# tuples.py

first = (1, 2, 3)
second = (4, 5, 6)

print "len(first) : ", len(first)
print "max(first) : ", max(first)
print "min(first) : ", min(first)
print "first + second :", first + second
print "first * 3 : ", first * 3
print "1 in first : ", 1 in first
print "5 not in second : ", 5 not in second
```
This example shows several basic operations with tuples. The `len()` function returns the number of elements in the first tuple. The `max()` function returns the maximum value, the `min()` minimum value. The addition operator adds two tuples, the multiplication operator multiplies the tuple. The `in` operator determines if the values is in the tuple.

```
$ ./tuples.py
len(first) :  3
```

```
max(first) :  3
min(first) :  1
first + second : (1, 2, 3, 4, 5, 6)
first * 3 :  (1, 2, 3, 1, 2, 3, 1, 2, 3)
1 in first :  True
5 not in second :  False
```
This is the output of the script.

Next will will do some indexing.

```
#!/usr/bin/python

# tuples2.py

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

print "five[0] : ", five[0]
print "five[-1] : ", five[-1]
print "five[-2] : ", five[-2]
print "five[:] : ", five[:]
print "five[0:4] : ", five[0:4]
print "five[1:2] : ", five[1:2]
print "five[:2] : ", five[:2]
print "five[:-1] : ", five[:-1]
print "five[:9] : ", five[:9]
```
To get a value from a tuple, we use square brackets `[]`. Note that we count indexes from 0. If there are five objects in a tuple, the indexes are 0...4. If we use a negative index, we get a value from the end of the tuple. So index -1 gets the last element, -2 gets the last but one element. Python enables to create *slices* from tuples. For this we use the `:` delimiter. For instance, `[0:4]` gives `(1, 2, 3, 4)`. Note that the last element is not included.

We can omit one or both indexes in a slice. The `[:4]` gives `(1, 2, 3, 4)`. It goes from the first element. The `[0:]` gives (1, 2, 3, 4, 5). This time, the last element is included. If we go out of bounds, we simply get all elements in the tuple.

```
$ ./tuples2.py
five[0] :  1
five[-1] :  5
five[-2] :  4
five[:] :  (1, 2, 3, 4, 5)
five[0:4] :  (1, 2, 3, 4)
five[1:2] :  (2,)
five[:2] :  (1, 2)
five[:-1] :  (1, 2, 3, 4)
five[:9] :  (1, 2, 3, 4, 5)
```
This is the output.

Tuples can contain several mix data types.

```
#!/usr/bin/python
```

```
# mix.py

mix = (1, 2, "solaris", (1, 2, 3))

print "mix[1] :", mix[1]
print "mix[2] :", mix[2]
print "mix[3] :", mix[3]
print "mix[3][0] :", mix[3][0]
print "mix[3][1] :", mix[3][1]
print "mix[3][2] :", mix[3][2]
```
In our example, we have put numbers, string and a tuple into the mix tuple.

```
$ ./mix.py
mix[1] : 2
mix[2] : solaris
mix[3] : (1, 2, 3)
mix[3][0] : 1
mix[3][1] : 2
mix[3][2] : 3
```
To get the elements from the nested tuple, we use two square brackets.

We have an exception when we work with tuples containing one element. Parentheses are also used in expressions. How do we distinguish between an expression and a one element tuple? The creators of the Python programming language decided to use a comma to denote that we are using a tuple.

```
#!/usr/bin/python

# onetuple.py

print (3 + 7)
print (3 + 7, )
```
In the first case we have and expression. We print number 10 to the console. In the second case we deal with a tuple. We print a tuple containing number 10.

```
$ ./onetuple.py
10
(10,)
```
Output.

Lists

A *list* is a mutable sequence data type. The list can contain mixed data types. A list and a tuple share many common features. Because a list is a modifiable data type, it has some additional operations. A whole chapter is dedicated to the Python list.

```
actors = ["Jack Nicholson", "Antony Hopkins", "Adrien Brody"]
```
The list is created using the square brackests `[]`.

```
#!/usr/bin/python
```

```
# list.py

num = [0, 2, 5, 4, 6, 7]

print num[0]
print num[2:]
print len(num)
print num + [8, 9]
```

As we have stated previously, we can use the same operations on lists as on tuples.

```
$ ./list.py
0
[5, 4, 6, 7]
6
[0, 2, 5, 4, 6, 7, 8, 9]
```

Output.

Next we will concentrate on additional operations we can do on lists. For example sorting.

```
#!/usr/bin/python

# sort.py

numbers = [4, 3, 6, 1, 2, 0, 5 ]

print numbers
numbers.sort()
print numbers
```

In our script we have a list of numbers. To sort those numbers, we use the built-in `sort()` function.

```
$ ./sort.py
[4, 3, 6, 1, 2, 0, 5]
[0, 1, 2, 3, 4, 5, 6]
```

The `reverse()` function will sort the elements of a list in reverse order.

```
numbers.reverse()    #  [5, 4, 3, 2, 1, 0]
```

Counting elements in a list is done with the `count()` method.

```
#!/usr/bin/python

# counting.py

numbers = [0, 0, 2, 3, 3, 3, 3]

print "zero is here ",  numbers.count(0), "times"
print "one is here ",   numbers.count(1), "times"
print "two is here ",   numbers.count(2), "time"
print "three is here ", numbers.count(3), "times"
```

The script counts number occurences in a list.

```
$ ./counting.py
```

```
zero is here  2 times
one is here  0 times
two is here  1 time
three is here  4 times
```

Next we will deal with inserting and deleting items from the list.

```
#!/usr/bin/python

# modify.py

names = []

names.append("Frank")
names.append("Alexis")
names.append("Erika")
names.append("Ludmila")

print names
names.insert(0, "Adriana")
print names
names.remove("Frank")
names.remove("Alexis")
del names[1]
print names
del names[0]
print names
```

In our example we first create an empty names list. We use the `append()` function to append new items to the list. The elements are appended in the consecutive way. The `insert()` function inserts new elements at a given position. The existing elements are not deleted, they are moved. The `rcodeove()` function removes a specific item from the list. If we want to delete an item based on the index, we use the `del` keyword.

```
$ ./modify.py
['Frank', 'Alexis', 'Erika', 'Ludmila']
['Adriana', 'Frank', 'Alexis', 'Erika', 'Ludmila']
['Adriana', 'Ludmila']
['Ludmila']
```

This is the output of the `modify.py` script.

There are other ways, how we can modify a list.

```
#!/usr/bin/python

# modify2.py

first = [1, 2, 3]
second = [4, 5, 6]

first.extend(second)
print first

first[0] = 11
first[1] = 22
```

```
first[2] = 33
print first

print first.pop(5)
print first
```
The `extend()` method appends a whole list to another list. To modify an element in a list, we can use the assignment operator. The `pop()` method takes an item from the list and returns it.

```
$ ./modify2.py
[1, 2, 3, 4, 5, 6]
[11, 22, 33, 4, 5, 6]
6
[11, 22, 33, 4, 5]
```
Output.

In the following example, we will find out indexes of elements.

```
#!/usr/bin/python

# index.py

numbers = [0, 1, 2, 3, 3, 4, 5]

print numbers.index(1)
print numbers.index(3)
```
To find an index in a list, we use the `index()` method. If there are more occurences of an element, the method returns the index of the first element.

```
$ ./index.py
1
3
```
Output of the `index.py` script.

Next we will do some transformations.

```
#!/usr/bin/python

# tuli.py

first = [1, 2, 3]
second = (4, 5, 6)

print tuple(first)
print list(second)

print first
print second
```
We can use `tuple()` function to create a tuple from a list and `list()` function to create a list from a tuple. Note that the original objects are not modified, the functions only return those transformed collections.

```
$ ./tuli.py
(1, 2, 3)
[4, 5, 6]
[1, 2, 3]
(4, 5, 6)
```

Sets

A `set` is an unordered collection of data with no duplicate elements. A set supports operations like union, intersection, or difference. Similar as in Mathematics.

```
#!/usr/bin/python

# sets.py

set1 = set(['a', 'b', 'c', 'c', 'd'])
set2 = set(['a', 'b', 'x', 'y', 'z'])

print "set1: " , set1
print "set2: " , set2
print "intersection: ", set1 & set2
print "union: ", set1 | set2
print "difference: ", set1 - set2
print "symmetric difference: ", set1 ^ set2
```

In our example we have two sets. We use the `set()` function to create sets. The intersection operation returns elements that are both in set1 and set2. The union operation returns all elements from both sets. The difference returns elements that are in the set1 but not is set2. And finally, the symmetric difference returns elements that are in set1 or set2, but not in both.

```
$ ./sets.py
set1:  set(['a', 'c', 'b', 'd'])
set2:  set(['a', 'x', 'b', 'y', 'z'])
intersection:  set(['a', 'b'])
union:  set(['a', 'c', 'b', 'd', 'y', 'x', 'z'])
difference:  set(['c', 'd'])
symmetric difference:  set(['c', 'd', 'y', 'x', 'z'])
```

This is the output of the `sets.py` script.

Next we introduce some other operations with sets.

```
#!/usr/bin/python

# sets2.py

set1 = set([1, 2])
set1.add(3)
set1.add(4)

set2 = set([1, 2, 3, 4, 6, 7, 8])
set2.remove(8)

print set1
print set2
```

```
print "Is set1 subset of set2 ? : ", set1.issubset(set2)
print "Is set1 superset of set2 ? : ", set1.issuperset(set2)

set1.clear()
print set1
```
The `add()` method adds an item to the set. The `rcodeove()` item removes an item from the set. The `clear()` method removes all items from the set. The `set1` is superset of `set2` if every element in `set2` is also in `set1`. The `set1` is a subset of `set2` if every element in `set1` is also in `set2`.

```
$ ./sets2.py
set([1, 2, 3, 4])
set([1, 2, 3, 4, 6, 7])
Is set1 subset of set2 ?  True
Is set1 superset of set2 ?  False
set([])
 fs = frozenset(['a', 'b', 'c'])
```
If we need an immutable set, we create a `frozenset`

Dictionaries

A Python *dictionary* is a group of key-value pairs. The elements in a dictionary are indexed by keys. Keys in a dictionary are required to be unique. Because of the importance of the dictionary data type, a whole chapter covers the dictionary in this Python tutorial.

```
#!/usr/bin/python

# dict.py

words = { 'girl': 'Maedchen', 'house': 'Haus', 'death': 'Tod' }

print words['house']

print words.keys()
print words.values()
print words.items()

print words.pop('girl')
print words
words.clear()
print words
```
Our first example shows some basic usage of the dictionary data type. We print a specific value, keys and values of the dictionary. The `items()` method returns a list of dictionarie's (key, value) pairs as tuples.

```
$ ./dict.py
Haus
['house', 'girl', 'death']
['Haus', 'Maedchen', 'Tod']
[('house', 'Haus'), ('girl', 'Maedchen'), ('death', 'Tod')]
Maedchen
{'house': 'Haus', 'death': 'Tod'}
```

{ }

In this part of the Python tutorial, we have described Python data types.