Python: Core Built-in Data Types

The purpose of this lecture is to cover the main built-in data types of the Python language. These data types are important since they constitute basic building blocks for several features of the language.

The core data types that are built-in Python are:

- Integers (`int`) and floats (`float`): store integers or real numbers;
- Strings (`string`): store characters and strings;
- Booleans (`bool`): store the values `True` or `False` and is used to test conditions;
- Sequences (`list`, `tuple`): list of elements where order matters;
- Dictionaries (`dict`): maps keys to values;

These data types combined with functions can be used to solve a huge number of problems. A full list of the built-in types are available on the built-in types documentation page. Python also has many built-in functions which we will cover as needed. For a complete list of the built-in functions, check the built-in functions documentation page.

1 Numbers

Numbers are separated into three different types: `int`, `float` and `complex`. Integers can be as big as required, as long as you have free memory available.

Variables in python are not strongly typed. This means the type of the variable is inferred at the time of declaration from its context. Thus, when creating numerical data, the type of variable will be inferred from its definition.

```
# integer
x = 3005
# big integer
y = 10000000000000000000000000000000000000000000000
z = 10**30000
```

We can print these numbers on the screen by using the function `print`.

```
print(x)
# can print multiple variables
print(x, y, z)
```

We can check the type of a variable with the built-in function `type`.

```
print(x, type(x))
print(y, type(y))
```

We can check the size of these integers (in bytes) with a helper function from the `sys` module, named `getsizeof`. We use functions from a module by importing the entire module with `import`. After importing the module, we use the dot notation to access functions from that module. For example, to use `getsizeof` from the module `sys`, we call `sys.getsizeof`.

```
# Import the module named sys
import sys
# Use the getsizeof function from sys to print size of variables:
print(sys.getsizeof(x), sys.getsizeof(y), sys.getsizeof(z))
```

The number of bytes assigned to the variable changes as the integer increases.

To create a float:

```
# Floating point numbers: floats
a = 3.1415
b = 2.718281828
```

Integers and floats support the usual mathematical operations:

```
x = 11
y = 2
# Addition, subtraction, multiplication and division
print(x + y, x - y, x*y, x/y)
# Integer division
print(x//y)
# Remainder of integer division
print(x%y)
```

We can also use the buil-in functions `abs` and `pow` to obtain absolute value and perform an exponentiation.

```
# Absolute value
print(abs(-x-y))
# Exponentiation
print(pow(x, 2))
print(x**2)                    # equivalent to pow
```

We can convert between integers and floats by calling the built-in functions `int` and `float`:

```
z = 3.2
print(int(z), type(int(z)))
h = 22
print(float(h), type(float(h)))
```

These functions are useful to convert strings into numbers.

2 Strings

There are several ways of creating strings:

```
# Create strings with single quotes '
first_name = 'John'
# or double quotes "
last_name = "Doe"
print(first_name, type(first_name))
# A string can also have quotation marks
```

```
more_strings = 'This is a string with "quotation marks"'
print(more_strings)
# To create strings with multiple lines use triple quotes '''
# or triple double quotes """
multiple_lines = '''This is a string
that spans
multiple lines. They can also be created with
triple quotation marks """ """'''
print(multiple_lines)
```

Strings are a collection of characters. Think of a vector, where each element is a letter. We can access letters in a string using the `[]` notation. For example, the first letter of a string has the index `0`, so we can access it using the syntax `[0]`:

```
name = "John"
print(name[0])
print(name[1], name[2])
# Last letter
print(name[3])
# Last letter of a string can be recovered with the index -1
print(name[-1])
```

We can use the built-in function `len` to get number of letters in a string.

```
print(len(name))
# Since index starts at 0, the last letter is at position len(name)-1
print(name[len(name) - 1])
# You get an error if you use invalid indices
print(name[len(name)])
```

Strings are immutable:

```
print(first_name[0])                    # first letter of string
first_name[0] = 'A'                     # error
```

We can concatenate strings with the `+` operator.

```
print(first_name + last_name)
print(first_name + ' ' + last_name)
```

Whatever is being concatenated has to have the type `str`, otherwise it needs to be converted to a string by using the built-in function `str`:

```
x = 1
print('Student ' + x + ': ' + first_name)  # error since x is not a string
# convert to string with str
print('Student ' + str(x) + ': ' + first_name)  # error since x is not a
    string
```

Everything in Python is an object. Objects hold values, like the value of the string, but can also hold other information, like properties and methods. Methods are functions which relate to the object itself. For example, a method might use the objects value to create a new object. In the case of strings, there is a method that takes the string in the variable and returns a new string with all letters capitalized:

```
# Capitalize string
print(first_name)
print(first_name.capitalize())
# All lower case
print(upper_case.lower())
```

String methods return another string, which can be stored in a variable:

```
upper_case = first_name.upper()
print(upper_case)
```

Remember that strings are immutable, so string methods all return new strings when used. We will use other methods after we talk about sequences and booleans. A complete list of the string methods is available on the string methods documentation page.

3 Booleans

There are two boolean values: `True` and `False`. The booleans are a subclass of integers, and, in some contexts, the integers 1 and 0 represent the booleans `True` and `False`.

We can obtain booleans via comparisons. There are 8 types of comparisons in Python:

```
print(1 < 1)
print(1 <= 1)
print(1 > 1)
print(1 >= 1)
print(1 == 1)
print(1 != 1)
print(1 is 1)  # compares if two object are the same
print(1 is not 1)
```

Booleans support the operations: `and`, `or` and `not`.

```
print((1 > 0) and (2 > 1))
print((1 > 2) or (1 < 2))
print((1 > 2) or (0 > 1))
print(not True)
```

The built-in function `bool` can convert any value to a boolean.

```
print(bool(0), bool(1))
print(bool('Guilherme'))
print(bool([]))
```

When we use the function `bool` Python evaluates whether the input of the function is associated with False or True. The most common objects that will lead to a `False` value are:

- Constants that are False by definition: `False` and `None`
- Numbers that are zero: 0, 0.0, 0j
- Empty sequences: `''`, `()`, `[]`, `{}`

Other objects that are non-empty will lead to a value of `True`.

4 Sequences

A `list` can hold any number of elements and types of objects:

```
list_of_numbers = [1, 2, 3, 2.3, 3.1, -1, 0, -5]
print(list_of_numbers)
list_of_strings = ['Guilherme', 'Salome', 'HFFE', 'Lecture 13']
mixed_list = [-1, False, 'Hello', 3.2]
```

Just like strings, elements of the list can be accessed by index using the `[]` notation. The index of a list starts at `0`.

```
print(list_of_numbers[0])
print(list_of_numbers[1])
```

The built-in function `len` gives the length of a list:

```
print(len(list_of_numbers))
print(len([]))
```

Because indices start at 0 and not at 1, the last index of a list is given by the length of such list minus 1. The last element of a list can also be accessed via the index `-1` (equivalent of Matlab's `end`):

```
print(mixed_list[3])
print(len(mixed_list))
print(mixed_list[len(mixed_list)])  # index out of range
print(mixed_list[len(mixed_list)-1])
print(mixed_list[-1])
```

We can recover several elements of a list, also known as slicing, using the `[]` notation.

```
numbers = [-1, 2, 5, 7, 9, 10, 12]
# recover first element
numbers[0]
# recover first three numbers
numbers[0:3]
# equivalent to
numbers[:3]
# recover from the third number to the fifth
numbers[2:5]
# recover from the 2nd number to the last
numbers[1:len(numbers)]
numbers[1:]
# recover the last number
numbers[-1]
```

We can also slice every other number using the notation `[start:stop:step]`:

```
# recover every other number
numbers[0:len(numbers):2]          # start:stop:step
numbers[0::2]                      # omission of stop means stop = len(
    numbers)
# recover every 3rd number
numbers[0::3]
```

Elements of a list can be modified:

```
list_of_numbers[0] = 1000
print(list_of_numbers)
list_of_numbers[-1] = 'oops'
print(list_of_numbers)
```

A list can be extended:

```
print(list_of_numbers)
list_of_numbers.append(100)     # adds to the end of list
print(list_of_numbers)
list_of_numbers.insert(0, 100)  # inserts at beginning of list
print(list_of_numbers)
```

```
list_of_numbers.insert(1, 100)  # inserts at position 1, shifts everythin
    to the right
```

It is important to note that adding elements to the beginning of a list or at a random location inside the list is slow. Appending at the end of the list is much faster.

A list can be shrunk.

```
print(list_of_numbers)
list_of_numbers.pop()           # removes last item
print(list_of_numbers)
list_of_numbers.pop(0)          # removes first item (this is slow)
print(list_of_numbers)
del list_of_numbers[1]          # removes element at index 1
print(list_of_numbers)
```

A `tuple` is also a sequence, but it is a sequence that cannot be modified after it is created.

```
tuple_of_numbers = (1,2,3,4,5)
print(tuple_of_numbers)
print(tuple_of_numbers[0], tuple_of_numbers[-1], len(tuple_of_numbers))
tuple_of_numbers[0] = 100       # raises error
tuple_of_numbers.append(100)    # raises error
```

Tuples are more efficient than lists since they occupy less bytes. Tuples are used when you want to fix a sequence of values that should not be changed.

A `range` is a special type of sequence. It constructs a sequence of numbers and is often used for looping a specific number of times in a loop.

```
print(list(range(0, 10)))
print(list(range(10)))
print(list(range(0, 10, 2)))
```

It is important to know that all functions that return a list of numbers in Python exclude the last number from the list. For example `range(5)` returns a list with the numbers `0,1,2,3,4`. This is because lists start in the index 0, so the last index of a list with 5 elements is the index 4. This contrasts with lists in Matlab which start at index 1, but is a small change that is more natural to programming and is the default in most programming languages.

5 Dictionaries

A mapping is an object that associates names (keys) to values (any object). The standard implementation of a mapping in Python is a dictionary: `dict`. Dictionaries are very efficient for searching keys and extracting the value associated with that key.

A dictionary can be created using curly brackets and passing a list of `key:value` pairs:

```
grades = {'A': 10, 'B': 8, 'C': 7, 'D': 8.5, 'E': 9.8}
print(type(grades))
print(len(grades))
grades['A']                     # returns value for grade of A
print(grades)
```

Dictionaries can be modified, its elements can be removed, new elements can be added, and existing elements can be updated.

```
# Change value of one key
grades['C'] = 7.5
print(grades)
# Change value of multiple keys
grades.update({'A': 0, 'B': 0})          # A and B were cheating
print(grades)
# Delete a value
del grades['B']                  # remove grade for B
print(grades)
# Extend dictionary
grades['F'] = 10                 # adds a new pair to the dictionary
print(grades)
```

We can recover (as a list) all of the keys of a dictionary, all of its values, and all of the `key:value` pairs:

```
list(grades.keys())              # a list of all dict keys
list(grades.values())            # a list of all dict values
list(grades.items())             # a list of the key:value pairs
```

Other methods are described here.