DIT DAKAR INSTITUTE OF TECHNOLOGY

"Introduction to Python programming."

February, 2021

Installation and configuration: Python3

To get started working with Python 3, you'll need to have access to the Python interpreter. Installing Python will depend on the Operating System you are using.

- If you're a **Windows user**, you'll need to:
  - Download the appropriate version of Python:
  - Navigate to the official Python website and go to the Download page.
  - Click on the link for the latest release - Python 3.x.x. (as of writing, the latest is Python 3.9.0)
  - Scroll to the bottom and select Windows x86-64 executable installer for 64-bit
  - Once the installer downloaded, run it by double-clicking on the downloaded file.
- For the **MacOS users**, the best way to install Python, is to:
  - install Homebrew, a MacOS package manager
  - then, install Python using Homebrew:
  - you'll need to open a terminal and type: brew install python3
- For most of the **GNU/Linux users**, Python3 comes pre-installed by default. If that is not your case, you may need to install the version 3 of Python:
  - grap a fresh version of Python: https://www.python.org/ftp/python/3.9.0/Python-3.9.0.tar.xz
  - extract the downloaded archive
  - open a terminal:
    - configure, sudo make, sudo make install

Once the installation is complete, you can run Python with the python3 command. If everything is okay, you should see the interpreter showing:

```
>> python3
Python 3.8.2 (default, Feb 28 2020, 07:36:25)
[GCC 7.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

We **strongly** advise you to install Python on your computer to follow this course. In the case you're not able to install Python on your computer, there are online interpreters available for you:

- www.python.org/shell
- pythonfiddle.com
- repl.it
- trinket.io

Operators:

Operators are special symbols that designate a computation, for example: addition and multiplication. Operators acts on operands. Operators are divided in categories, which are:

Arithmetic operators:

We use them with numeric values so we can perform common mathematical operations:

```
+     Addition        x + y
-     Subtraction     x - y
*     Multiplication  x * y
/     Division        x / y
%     Modulus         x % y
**    Exponentiation  x ** y
//    Floor division  x // y
```

Assignment operators:

Used to assign values to variables:

```
=     x = 5           x = 5
+=    x += 3          x = x + 3
-=    x -= 3          x = x - 3
*=    x *= 3          x = x * 3
/=    x /= 3          x = x / 3
%=    x %= 3          x = x % 3
//=   x //= 3         x = x // 3
**=   x **= 3         x = x ** 3
```

Comparison operators:

Used to compare two values:

```
==    Equal                     x == y
!=    Not equal                 x != y
>     Greater than              x > y
<     Less than                 x < y
>=    Greater than or equal to  x >= y
<=    Less than or equal to     x <= y
```

Logical operators:

Used to combine conditional statements: (We'll come back later to what's conditional statements)

```
and    Returns True if both statements are true                  x < 5 and  x < 10
or     Returns True if one of the statements is true             x < 5 or x < 4
not    Reverse the result, returns False if the result is true   not(x < 5 and x < 10)
```

Identity operators:

Used to compare objects, not if they are equal, but if they are the same object, with the same memory location:

```
is      Returns true if both variables are the same object      x is y
is not  Returns true if both variables are not the same object  x is not y
```

Membership operators:

Used to test if a sequence is presented in an object:

```
in      Returns True if an object x is present in the object y         x in y
not in  Returns True if an object x is not present in the object y    x not in y
```

Values and types

In Python, everything is an object. In all your programs, what you manipulate are objects. To manipulate all those objects, you need: - to name them, using "variables", a variable references an object. - to know their types, every object has a type.

The values 100, 250, and "Hello World!", belong to different types, for example, 2 is an integer, and "Hello, World!"" is a string, *notice the double quotes*. If you are not sure what type a value has, you can use the **type** function.

There are standard types that are built into the Python interpreter:

- Numbers: int, float, complex
- Booleans: True, False
- Sequences:
  - immutable: Strings, Tuples, Bytes
  - mutable: Lists, ByteArrays
- Set types: Sets, Frozen sets
- Mappings: Dictionaries

Variables, expressions and statements:

Python programs can be decomposed into **modules**. A module can be decomposed in **statements**. A statement can be decomposed in **expressions**. An expression can be decomposed in **identifiers, literals, and operators (objects)**.

```
>>> # A statement is anything that can make up one or several lines of Python code.
>>> # An expression is part of a statement, and should produce at least one value.
>>> # Example:
>>> x = 1 # a statement
>>> x + 2 # an expression
```

Here is an assignment statements that creates new variable_name_ and gives it the value *"foo"*:

```
>>> name = "foo"
```

A variable name can contain both letters and numbers, but they have to **begin with a letter**. It is legal to use uppercase letters, but the convention asks us to begin variable names with a lowercase letter.

Common strings operations:

Strings are bits of text. They can be defined as anything between double quotes. Python gives us some operations to manipulate our strings. The most common operations to do on strings are:

- length
- indexing
- slices
- in operator
- concatenation

But there is also interesting functions like:

- capitalize
- title
- upper
- lower
- find
- center
- format
- join
- etc ...

Functions:

In mathematics, a function is a relationship or mapping between one or more inputs and a set of outputs. In programming, a function is a unit of computation, a block of code, that encapsulate a specific task.

Functions are a way for us to make a program smaller by eliminating repetitive code. We use functions to bundle a set of instructions that we want to reuse.

Example:
We already had the chance to manipulate some functions: **print, type, help, dir, len, abs, int**, ... The language has a number of functions that are always available for us.

User defined functions:

It is also possible to define our own functions. The syntax for defining a Python function is:

```
  def <function_name>([<parameters>]):
      <statement(s)>
```

```
def              The keyword that informs Python that a function is being defined
<function_name>  A valid Python identifier to name the function
<parameters>     An optional, comma-separated list of parameters that may be passed to the function
:                Punctuation that denotes the end of the Python function header
<statement(s)>    A block of valid Python statements
```

Examples:

```
    # a function that does nothing, notice the pass keyword
    def foo():
        pass

    # a function that takes no arguments and returns nothing
    def bar():
        print("hello world")
```

```
# a function that takes one argument
# prints the passed parameter four times
# and returns nothing
def print4(msg):
    for _ in range(4):
        print(msg)

# a function that takes no arguments
# and always return an integer, 42
def bar():
    return 42

# a function that takes 3 arguments
# and return a sum of the passed parameters
def baz(a, b, c):
    s = a + b + c
    return s

# return cause l'arrêt immédiat de la fonction
# cette fonction prend un parametre positionnel: x
def f(x):
    if x < 0:
        return
    if x > 100:
        return
    print(x)


# une fonction qui prend 3 paramètres positionnels (ou paramètres requis)
# et renvoie la somme de ces 3 paramètres
def baz(a, b, c):
    s = a + b + c
    return s


# une fonction qui prend 5 parametres
# 3 paramètres positionnels: a, b, c
# 2 paramètres optionnels: d, e
# d et e ont des "valeurs par défaut"
def qux(a, b, c, d="foo", e=42):
    print(d, e)
    s = a + c + b
    return s

# on retiendra que les paramètres positionnels
# arrivent toujours avant les paramètres optionnels
# autrement on obtient une erreur
>>> def qux( d="foo", e=42, a, b, c):
...     print(d, e)
...     s = a + c + b
...     return s
...
  File "<stdin>", line 1
SyntaxError: non-default argument follows default argument
>>>
```

The syntax for calling a function is:

```
<function_name>([<arguments>])
```

```
<arguments> are the values passed into the function.
<arguments> correspond to the <parameters> in the Python function definition.
```

Both a function definition and a function call must always include parentheses, even if they're empty.

Control flow statements:

The if statement:

You will often need to execute some statements only **if some condition holds**, or choose to execute some code depending on several mutually exclusive conditions. The Python statement if, lets you conditionally execute blocks of statements. Here's the syntax for the if statement:

```
if x < 0:
    print("x is negative")
elif x % 2:
    print("x is positive and odd")
else:
    print("x is even and non-negative")
```

The while statement:

The while statement supports repeated execution of block of statements that is controlled by a conditional expression. The loop body should contain code that eventually makes the loop condition false, or the loop will never end unless an exception is raised or the loop body executes a break statement. Here's an example:

```
number = 23
running = True

# running it true, so we enter the while loop
while running:
    guess = int(input('Please, enter an integer, : '))

    if guess == number:
        print('Congratulations, you guessed it.')
        # this causes the while loop to stop
        running = False
    elif guess < number:
        print('No, it is a little higher than that.')
    else:
        print('No, it is a little lower than that.')
else:
    # A while statement can have an optional else clause.
    print('The while loop is over.')
    # Do anything else you want to do here

print('Done')
```

The break statement is used to stop the execution of a looping statement, even if the loop condition has not become False. Example:

```
while True:
    s = input('Enter something : ')
    #
    if s == 'quit':
        break
    print('Length of the string is', len(s))
print('Done')
```

The for statement:

The for statement in Python supports repeated execution of statements that is controlled by an iterable expression. Here's an example for the for statement:

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

```
>>> # Measure some strings:
... words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

for statements may have an else clause; it is executed when the loop terminates through exhaustion of the iterable. The else clause executes after the loop completes normally, example:

```
for i in mylist:
   if i == theflag:
       break
   process(i)
else:
    raise ValueError("List argument missing terminal flag.")
```

Data structures:

Python built-in data structures are a way for you to perform more complex tasks.

Data structures are the fundamental constructs around which you build more interesting programs. Data structures are *structures which can hold some datas together*. In other words, data structures are used to store a collection of related datas.

There are four built-in data structures in Python: **list, tuple, dictionary and set**. Let's see how to use each of them:

Lists:

- List literals are written within square brackets [ ].
- Lists can hold arbitrary types of elements.
- Lists work similarly to strings, we use the len() function to get the size and square brackets [ ] to access data.
- The first element always at index 0.

Once you have created a list, you can add, remove or search for items in the list. Since we can add and remove items, we say that a list is a **mutable** data type: the type can be altered.

```
>>> # Example:
>>> arr = ["one", "two", "three"]
>>> arr[0]
'one'

>>> # Lists have a nice repr:
>>> arr
['one', 'two', 'three']

>>> # Lists are mutable:
>>> arr[1] = "hello"
>>> arr
['one', 'hello', 'three']

>>> del(arr[1])
>>> arr
['one', 'three']

>>> # Lists can hold arbitrary data types:
>>> arr.append(23)
>>> arr
['one', 'three', 23]
```

Tuples:

Tuples are data structures for ""grouping" objects. Tuples are another data structure that can hold elements of arbitrary data types. However, tuple objects are **immutable**. Elements can't be added or removed dynamically.

```
>>> # Example:
>>> arr = ("one", "two", "three")
>>> arr[0]
'one'

>>> # Tuples have a nice representation:
>>> arr
('one', 'two', 'three')

>>> # Tuples are immutable:
>>> arr[1] = "hello"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment

>>> del(arr[1])
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object doesn't support item deletion

>>> # Tuples can hold arbitrary data types:
>>> # (Adding elements creates a copy of the tuple)
>>> arr + (23,)
('one', 'two', 'three', 23)

>>> # Notice the difference of identity
>>> id(_)
>>> 139687616067920
>>> id(arr)
>>> 139687614530944
```

Dictionaries:

Dictionaries, one of the most important and frequently used data structures in computer science. Dictionaries store an arbitrary number of objects, each identified by a unique key. They're so important that the standard library includes:

- plain dictionary objects
- and a number of **specialized dictionary** implementations.

```
>>> # Example:
>>> car2 = {
...     "color": "blue",
...     "mileage": 40231,
...     "automatic": False,
... }

>>> # Dicts have a nice repr:
>>> car2
{'color': 'blue', 'automatic': False, 'mileage': 40231}

>>> # Get mileage:
>>> car2["mileage"]
40231

>>> # Get mileage:
>>> car2["automatic"]
False

>>> # Dicts are mutable:
>>> car2["mileage"] = 12
>>> car2["windshield"] = "broken"
>>> car2
{'windshield': 'broken', 'color': 'blue',
 'automatic': False, 'mileage': 12}
```

Sets:

Sets are unordered collections of simple objects that doesn't allow duplicate elements. Sets are used to quickly test a value for membership in the set, to insert or delete new values from a set, and to compute the union or

intersection of two sets. A common use of sets in Python is computing standard math operations such as **union, intersection, difference, and symmetric difference**.

```
>>> # Example:
>>> # Be careful: To create an empty set you'll need to call the set() constructor.
>>> # Using empty curly-braces {} will create an empty dictionary instead.
>>> vowels = {"a", "e", "i", "o", "u"}
>>> "e" in vowels
True

>>> letters = set("alice")
>>> letters.intersection(vowels)
{'a', 'e', 'i'}

>>> vowels.add("x")
>>> vowels
{'i', 'a', 'u', 'o', 'x', 'e'}

>>> len(vowels)
6
```