Fundamentals of Information Systems

Python Programming (for Data Science)

Master's Degree in Data Science

Gabriele Tolomei
gtolomei@math.unipd.it
University of Padua, Italy
2018/2019
October 15, 2018

Lecture 2: Python Language Basics

Language Syntax

Indentation rather than Braces

- Python uses whitespaces (tabs or spaces) to structure code instead of using braces `{ }` as in many other languages like R, C++, Java, and Perl.

In [ ]:

```
"""A colon ':' denotes the start of an indented code block after which all of the code
must be indented by the same amount of whitespaces until the end of the block.
"""
for x in array:
    if x < 0:
        print("x = {} is strictly negative".format(x))
    elif x > 0:
        print("x = {} is strictly positive".format(x))
    else:
        print("x is {}".format(x))
```

Note: *We strongly recommend that you use **4 spaces** as your default indentation and that your editor replace tabs with 4 spaces. Many text editors have a setting that will replace tab stops with spaces automatically (do this!).*

No Need for Semicolons

- Python statements **do not** need to be terminated by semicolon "`;`".
- Unless you want to separate multiple statements on the same line:

```
x = 3; y = 4; z = 5
```

- Putting multiple statements on one line is generally discouraged, as it makes code less readable.

Comments

- Any text preceded by the hash mark (pound sign) "`#`" is ignored by the Python interpreter.
- This is often used to add comments to code or to (temporarily) exclude certain blocks of code without deleting them.
- Comments spawning across multiple lines need to be included within `"""MULTILINE COMMENT HERE"""`.

In [ ]:

```
results = [] # This is an inline comment
for line in file_handle:
    # This is a comment
    """This is a multiline
    comment
    """
    '''This is
    also a multiline comment
    '''
    # The following two lines of code are excluded from execution
    # if len(line) == 0:
    #   continue
    results.append(line.replace('foo', 'bar'))
```

Variables

- There are many cases where *values* (being they strings, numbers, etc.) should be (temporarily) "saved" into *variables*.
- In a nutshell, a variable is just the **name** of the container for a **value**.
- A variable allows us to refer to the same value (possibly multiple times in our code) without having to explicitly write such a value down.

Naming Rules (1 of 2)

- Variable names can only contain **letters**, **numbers**, and **underscores**.
- Variable names can start with a letter or an underscore, but **cannot start with a number**.
- Spaces are not allowed, so we use underscores instead of spaces. For example, use `student_name` instead of `student name`.

Naming Rules (2 of 2)

- Variable names cannot be Python keywords (e.g., `for`, `import`, `from`, etc.).
- Variable names should be descriptive, without being too long.
- Be careful about using the lowercase letter `l` and the uppercase letter `O` where they could be confused with the numbers `1` (one) and `0` (zero).

Python's Object Model

Everything is an Object

- Every number, string, data structure, function, class, module, and so on exists in the Python interpreter in its own "box", which is referred to as a Python **object**.
- Each object has an associated **type** (e.g., string or function) and internal **data**.
- This makes the language very *flexible*, as even functions can be treated like any other object.

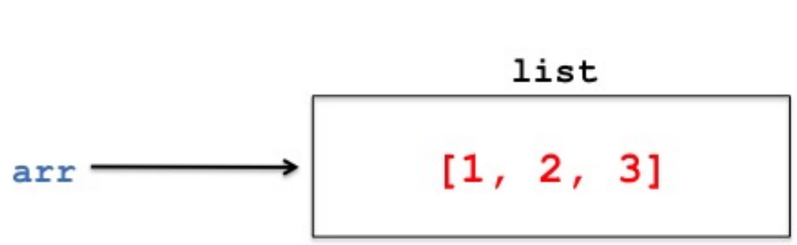
Variables as References

- When you assign a **variable** (or name) in Python, you are in fact creating a **reference** to the object on the right hand side of the assignment expression.

```
# The variable named 'arr' is actually a reference
# to the list object [1, 2, 3]
arr = [1, 2, 3]
```

Variables as References

- Assignment is also referred to as **binding**, as we are binding a name to an object.
- Variable names that have been assigned may occasionally be referred to as *bound variables*.



Value vs. Reference

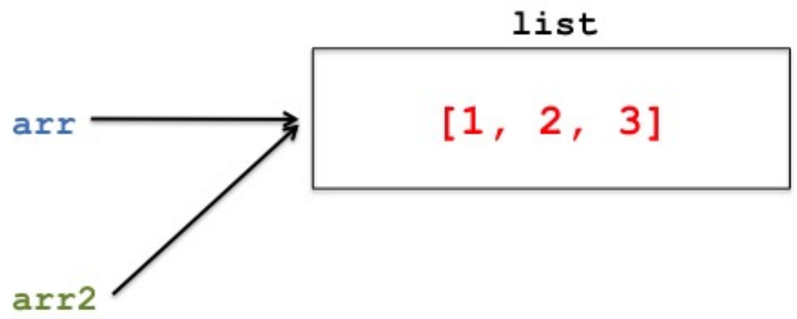
- Suppose we assign (the value referenced by) `arr` to another variable `arr2`.

```
arr2 = arr
```

- In some languages, this assignment would cause the data referenced by `arr` (i.e., the list `[1, 2, 3]`) to be copied.

Value vs. Reference

- In Python, `arr` and `arr2` now actually refer to the *same* object, namely the original list.

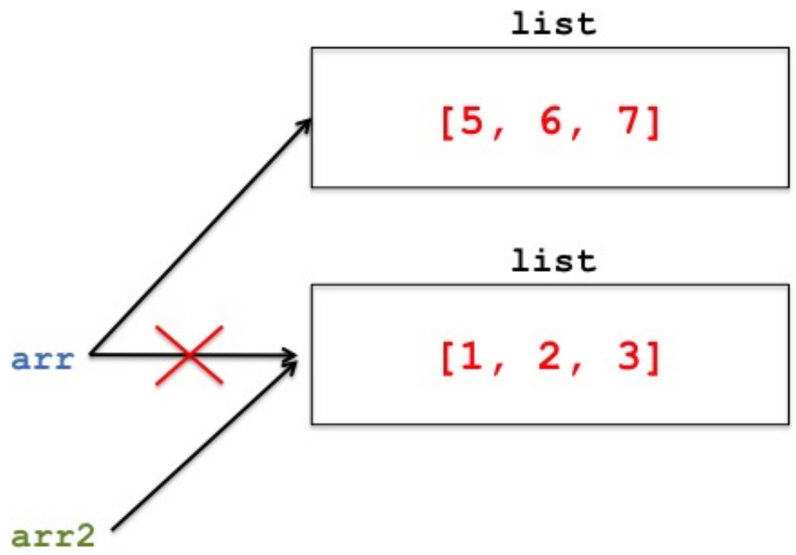


In [1]:

```python
# Assign the list object to the variable named arr
arr = [1, 2, 3]
# Print this variable
print("The value referenced by arr is: {}".format(arr))
# Assign arr to another variable arr2
# (i.e., make arr2 point to the same object pointed by arr)
arr2 = arr
# Print arr2 so as to check the value printed out is actually the same of arr
print("The value referenced by arr2 is: {}".format(arr2))
# Modify arr by appending a new element to the original list
arr.append(4)
# Print arr2 to see if it is affected too
print("After modifying the value referenced by arr, arr2 points to: {}".format(arr2))
# Note that arr2 is NOT affected if arr is re-assigned (i.e., re-bound)
# to a different object!
arr = [5, 6, 7]
# Now arr is re-bound to a new list object
print("After rebinding arr, the value it references now is: {}".format(arr))
print("After rebinding arr, the value referenced by arr2 is: {}".format(arr2))
```

```
The value referenced by arr is: [1, 2, 3]
The value referenced by arr2 is: [1, 2, 3]
After modifying the value referenced by arr, arr2 points to: [1, 2, 3, 4]
After rebinding arr, the value it references now is: [5, 6, 7]
After rebinding arr, the value referenced by arr2 is: [1, 2, 3, 4]
```

Re-binding Variables



Object's Attributes and Methods

- Objects in Python typically have both **attributes** and **methods**.
    - Attributes are other Python objects stored "inside" the object and representing its *internal state*;
    - Methods are functions associated with an object which can have access to/manipulate the object's internal state.
- Both of them are accessed via the syntax `obj.attribute_name` or `obj.method_name(args...)` where `(args...)` are the input arguments of the method.

In [ ]:

```
# The list of available attributes/methods of an object
# can be found by typing obj.<TAB>
arr.
```

Object's Attributes and Methods

- In the previous example:

```
# arr is a list object;
# append is invoked to insert a new element
arr.append(4)
```

Mutable vs. Immutable Objects

- Many objects in Python are **mutable**, such as lists, dictionaries, sets, or most user-defined types (classes).
- This means that the object or values that they contain can be modified.
- Others, like integers, strings, and tuples are **immutable**.

In [2]:

```
# Defining a (mutable) list object
a_list = [1, 2, 3]
print(a_list)
# Modify the content of the object referenced by a_list
a_list[1] = True
print(a_list)
```

```
[1, 2, 3]
[1, True, 3]
```

In [3]:

```
# Defining a string object (immutable)
a_string = 'This is an immutable string'
print(a_string)
# Try to modify the object referenced by a_string
a_string[0] = 't'
print(a_string)
```

```
This is an immutable string

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-3-f5237db9f438> in <module>()
      3 print(a_string)
      4 # Try to modify the object referenced by a_string
----> 5 a_string[0] = 't'
      6 print(a_string)

TypeError: 'str' object does not support item assignment
```

In [4]:

```python
# Defining a (mutable) list object
a_list = [1, 2, 3]
# make `another_list` reference to the same object referenced by `a_list`
another_list = a_list
print("a_list: {}".format(a_list))
print("another_list: {}".format(another_list))
# check if the two (symbolic) names actually refer to the same object
print("a_list (id): {}".format(id(a_list)))
print("another_list (id): {}".format(id(another_list)))
# Modify the content of the object referenced by a_list
a_list += [4, 5] # the same as a_list.extend([4, 5])
print("a_list: {}".format(a_list))
print("another_list: {}".format(another_list))
print("a_list (id): {}".format(id(a_list)))
print("another_list (id): {}".format(id(another_list)))
```

```
a_list: [1, 2, 3]
another_list: [1, 2, 3]
a_list (id): 4495929672
another_list (id): 4495929672
a_list: [1, 2, 3, 4, 5]
another_list: [1, 2, 3, 4, 5]
a_list (id): 4495929672
another_list (id): 4495929672
```

In [5]:

```
# Defining an (immutable) integer object
x = 42
# make `y` reference to the same object referenced by `x`
y = x
print("x: {}".format(x))
print("y: {}".format(y))
# check if the two (symbolic) names actually refer to the same object
print("x (id): {}".format(id(x)))
print("y (id): {}".format(id(y)))
# Modify the content of the object referenced by x
x += 5 # 42 can't be mutated, a NEW object is created here (and make it referenced by x)
print("x: {}".format(x))
print("y: {}".format(y))
print("x (id): {}".format(id(x)))
print("y (id): {}".format(id(y)))
```

```
x: 42
y: 42
x (id): 4447148656
y (id): 4447148656
x: 47
y: 42
x (id): 4447148816
y (id): 4447148656
```

In [8]:

```
# Defining a (mutable) list object containing heterogeneous
# and possibly immutable elements
a_list = [1, 'foo', [2,3], (4,5)]
# What will you expect if we modify the 2nd element of the list?
a_list[1] = 'bar'
print(a_list)
```

```
[1, 'bar', [2, 3], (4, 5)]
```

In [9]:

```
# Defining a (mutable) list object containing heterogeneous
# and possibly immutable elements
a_list = [1, 'foo', [2,3], (4,5)]
# What will you expect if we modify the 2nd element of the list?
a_list[1] = 'bar'
print(a_list)
# What if, instead, we try the following:
a_list[1][0] = 'z'
```

```
[1, 'bar', [2, 3], (4, 5)]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-9-77836e70ab79> in <module>()
      6 print(a_list)
      7 # What if, instead, we try the following:
----> 8 a_list[1][0] = 'z'

TypeError: 'str' object does not support item assignment
```

Function Call (1 of 2)

- Functions are called using parentheses and passing zero or more arguments.
- Optionally, the returned value can be assigned to a variable:

```
foo()
result = bar(a, b)
```

Function Call (2 of 2)

- Functions can take both *positional* and *keyword* arguments:

```
result = bar(a, b, c=42, d='baz')
```

Passing Arguments into a Function

- Arguments are *passed by assignment*. The rationale behind this is twofold:
    - The parameter passed in is actually a **reference** to an object (*but the reference itself is* ***passed by value***);
    - As we have already seen, some data types are mutable, but others aren't.

Passing *Mutable* Arguments into a Function

- If you pass a ***mutable*** object into a function (method), the function gets a *reference* to that same object.
- Within the function's body you can modify the referenced object as you like, and any change to it is also "visible" outside the method (*side effects*).
- Instead, if you rebind the reference in the function's body, the outer scope will know nothing about it, and after the function returns the outer reference will still point at the original object.

In [11]:

```
'''
This example shows that changes to a mutable object (i.e., a list)
inside a function are also reflected outside of it.
'''
def try_to_change_list_content(a_list):
    print('Input list received by the function: ', a_list)
    a_list.append(4)
    print('Modified list by the function: ', a_list)

input_list = [1, 2, 3]

print('Before function call, the list is: ', input_list)
try_to_change_list_content(input_list)
print('After function call, the list is: ', input_list)

```

```
Before function call, the list is:  [1, 2, 3]
Input list received by the function:  [1, 2, 3]
Modified list by the function:  [1, 2, 3, 4]
After function call, the list is:  [1, 2, 3, 4]
```

In [10]:

```
'''
This example shows that rebinding the reference to a mutable object (i.e., a list)
inside a function does NOT rebind the outer reference.
'''
def try_to_change_list_reference(a_list):
    print('Input list received by the function: ', a_list)
    a_list = [5, 6, 7]
    print('Rebind list by the function to: ', a_list)

input_list = [1, 2, 3]

print('Before function call, the list is: ', input_list)
try_to_change_list_reference(input_list)
print('After function call, the list is: ', input_list)
```

```
Before function call, the list is:  [1, 2, 3]
Input list received by the function:  [1, 2, 3]
Rebind list by the function to:  [5, 6, 7]
After function call, the list is:  [1, 2, 3]
```

Passing *Immutable* Arguments into a Function

- If you pass an ***immutable*** object to a method, you still can't rebind the outer reference **and** you can't even modify the object.

In [12]:

```
'''
This example shows that any attempt of change to an immutable object (i.e., a string)
inside a function cannot be performed.
'''
def try_to_change_string_content(a_string):
    print('Input string received by the function: ', a_string)
    a_string[2] = 'z'
    print('Modified string by the function: ', a_string)

input_string = 'Bar'

print('Before function call, the string is: ', input_string)
try_to_change_string_content(input_string)
print('After function call, the string is: ', input_string)
```

```
Before function call, the string is:  Bar
Input string received by the function:  Bar

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-12-fc30c4ce7f4a> in <module>()
     11
     12 print('Before function call, the string is: ', input_string)
---> 13 try_to_change_string_content(input_string)
     14 print('After function call, the string is: ', input_string)

<ipython-input-12-fc30c4ce7f4a> in try_to_change_string_content(a_string)
      5 def try_to_change_string_content(a_string):
      6     print('Input string received by the function: ', a_string)
----> 7     a_string[2] = 'z'
      8     print('Modified string by the function: ', a_string)
      9

TypeError: 'str' object does not support item assignment
```

In [13]:

```
'''
This example shows that rebinding the reference to an immutable object (i.e., a string)
inside a function does not rebind the outer reference.
'''
def try_to_change_string_reference(a_string):
    print('Input string received by the function: ', a_string)
    a_string = 'Ciao Mondo!'
    print('Rebind string by the function to: ', a_string)

input_string = 'Hello World!'

print('Before function call, the string is', input_string)
try_to_change_string_reference(input_string)
print('After function call, the string is', input_string)
```

```
Before function call, the string is Hello World!
Input string received by the function:  Hello World!
Rebind string by the function to:  Ciao Mondo!
After function call, the string is Hello World!
```

Dynamic References, Strong Types

- In contrast with many **statically-typed** languages, such as Java and C++, object references in Python have **no type** associated with them.
- A language is statically typed if the type of a variable must be known at compile time (i.e., the programmer has to specify the type of the variables she declares).

```java
/* A variable definition in Java.
The programmer needs to explicitly inform the compiler
about its type (String) at this stage. */
String x = new String("Hello World!");

# A variable definition in Python.
# No information about its type is needed.
x = 'Hello World!'
# The same name can be rebound to a different type.
x = 5
```

In [14]:

```
# 1. Assign x to a reference to a string object
x = 'foo'
# 2. Verify the type associated with x
print(type(x))
# 3. Rebind x to a reference to an integer object
x = 5
# 4. Verify the (new) type associated with x
print(type(x))
```

```
<class 'str'>
<class 'int'>
```

Dynamic References, Strong Types

- Variables are just names (identifiers) for objects within a particular namespace.
- The type information is stored in the object itself and can be inferred at runtime.
- **Note:** You might be tempted to conclude that Python is not a "typed language". **This is not true!**

In [15]:
```
# Let's see what happens if we try to sum a string and an integer
'7' + 7
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-15-543e7ccd630f> in <module>()
      1 # Let's see what happens if we try to sum a string and an integer
----> 2 '7' + 7

TypeError: must be str, not int
```

Dynamic References, Strong Types

- In some languages, such as Visual Basic, the string `'7'` might get implicitly converted (or casted) to an integer, thus yielding `14`.
- Yet in other languages, such as JavaScript, the integer `7` might be casted to a string, yielding the concatenated string `'77'`.
- Python is considered a **strongly-typed** language, which means that every object has a specific type (or *class*), and implicit conversions will occur only in some circumstances.

In [16]:
```
# Use the isinstance() method to check the type associated with an object.
x = 7
# The method takes as input a tuple of types to test against of.
# It returns true if there exists at least one type in the tuple
# which corresponds to the correct type.
isinstance(x, (int, list))
```

Out[16]: True

Structuring Your Python Code

Importing Modules

- In Python a ***module*** is simply a `.py` file containing function and variable definitions along with such things imported from other `.py` files.

In [17]:

```
# Consider the following code snippet is contained in a file named 'my_module.py'
# Define a constant
PI = 3.14159

# Define function foo
def foo(x):
    return x * 2

# Define function bar
def bar(a, b):
    return a - b
```

In [ ]:

```
# If we want to access the variables and functions defined in 'my_module.py'
# from another file in the SAME directory we could do as follows
import my_module
result = my_module.foo(5)
pi = my_module.PI

# Or, equivalently
from my_module import foo, bar, PI
result = bar(42, PI)

# Finally, using the 'as' keyword you can give imports different variable names
import my_module as mm
from my_module import PI as pi, bar as g

x = mm.foo(pi)
y = g(6, pi)
```

Module Search Path

- In the example above, we show how a Python module called `my_module` can be imported to another Python file in the **same** directory.
- When `my_module` is imported (from another Python file) the interpreter searches for it as follows:

1. First, it searches for a built-in module with that name (`my_module`)
2. If no standard module is found, it then searches for a file named `my_module.py` in a list of directories given by the variable `sys.path`.

Module Search Path: `sys.path`

- `sys.path` is initialized from these locations:
    - The directory containing the input script (or the current directory when no file is specified);
    - `PYTHONPATH` environment variable (i.e., a list of directory names, with the same syntax as the shell variable `PATH`);
    - The installation-dependent default.
- More information on Python modules can be found here.

Summary

- Python language basic syntax:
    - indentation rather than braces!
- Object model:
    - variables as *references* to object
    - dynamic binding/typing
    - strongly-typed
- Python modules