

Fundamentals of Machine Learning Python for Data Science – Part 1

ML Instructional Team

08 April, 2020



The Python Interpreter

- Python is an interpreted language (as opposed to compiled languages such as C++).
- Commands can be executed interactively, or a series of commands can be saved in a plain text file known as source code or a script.
- For Python, source code is normally stored in a file with the .py suffix (e.g., hello.py).
- We will use Python 3 for this class.

How to run Python code (1)

We can use one of the following ways

- Start Python at the command line by typing
`$ python`
- Edit `source.py` and run
`$ python source.py`
- Make the python file executable and run it directly
 - Make the Python source code executable
`$ chmod +x source.py`
 - Put the path to the Python interpreter at the very first line of the Python source code, e.g.
`#!/usr/local/bin/python`
 - Run the Python source code directly
`$./source.py`

How to run Python code (2)

- Use an IDE (Integrated Development Environment) such as Pycharm.
- Use Jupyter notebooks
- Use online notebooks such as
 - <https://colab.research.google.com/>
 - <https://www.kaggle.com/>

Sample Python source code

```
def calc_mean(f_list):  
    l_len = len(f_list)  
    # Calculate the mean  
    l_sum = 0  
    for e in f_list:  
        l_sum += e  
    l_mean = float(l_sum) / l_len  
    return l_mean
```

```
my_list = [4, 5, 20]  
calc_mean(my_list)
```

Identifiers, Objects, and the Assignment Statement

- Python is an object-oriented language and all data types are based on classes.
- Consider the assignment (Example from [1] Goldwasser, Goodrich, and Tamassia)

```
temperature = 98.6
```

- This associates the identifier *temperature* with a floating-point object with value 98.6.
- Identifiers in Python are case-sensitive, so *temperature* and *Temperature* are distinct names.
- Identifiers can be composed of almost any combination of letters, numerals, and underscore characters.
- An identifier, however, cannot begin with a numeral (E.g., '4runner' is not a valid identifier).

```
temperature = 98.6
```

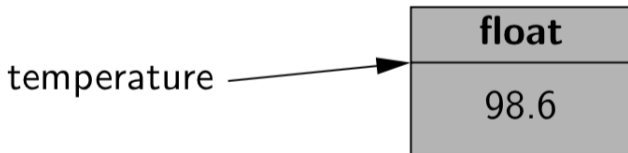


Figure: The identifier *temperature* is associated with a floating-point object with value 98.6 (Figure from [1])

Identifiers, Objects, and the Assignment Statement

- An identifier cannot be one of the reserved words (Table from [1] Goldwasser, Goodrich, and Tamassia)

| Reserved Words | | | | | | | | |
|----------------|--------|----------|---------|--------|----------|-------|--------|-------|
| False | as | continue | else | from | in | not | return | yield |
| None | assert | def | except | global | is | or | try | |
| True | break | del | finally | if | lambda | pass | while | |
| and | class | elif | for | import | nonlocal | raise | with | |

Table 1.1: A listing of the reserved words in Python. These names cannot be used as identifiers.

- Unlike statically typed languages such as C++, Python is a dynamically typed language.
- We do not need to associate an identifier with a particular data type during declaration.
- We can associate an identifier with any type of object, and later reassign it to another object of the same (or different) type.
- Although an identifier has no declared type, the object to which it refers has a definite type.

```
original = temperature
```

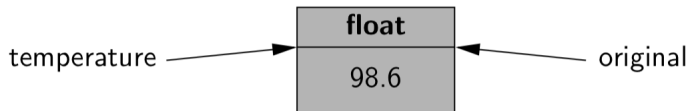


Figure: We can assign a second identifier, *original*, to the floating-point object. Now both *temperature* and *original* are **aliases** of the floating-point object (Figure from [1])

- Any alias of an object can be used to access the object.
- If an object supports operations to change its state, changes made using an alias (without using an assignment statement) will still be effective as seen from another alias
- If we assign a new value to an alias using an assignment statement, the alias will point to a new object

```
temperature = temperature + 5.0
```

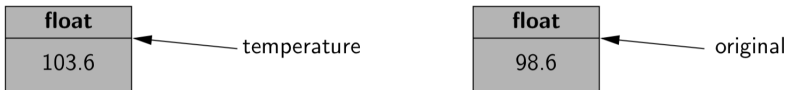


Figure: If we assign a new value to an alias using an assignment statement, the alias will point to a new object (Figure from [1])

```
[ ] group = ['An', 'Binh', 'Cuong', 'Dai']
```

```
[ ] group_2 = group
```

```
[ ] group_2
```

```
↳ ['An', 'Binh', 'Cuong', 'Dai']
```

```
[ ] group_2.append('En')
```

```
[ ] group_2
```

```
↳ ['An', 'Binh', 'Cuong', 'Dai', 'En']
```

```
[ ] group
```

```
↳ ['An', 'Binh', 'Cuong', 'Dai', 'En']
```

Figure: If an object supports operations to change its state, changes made using an alias (without using an assignment statement) will still be effective as seen from another alias

```
[ ] group_2
```

```
↳ ['An', 'Binh', 'Cuong', 'Dai', 'En']
```

```
[ ] group
```

```
↳ ['An', 'Binh', 'Cuong', 'Dai', 'En']
```

```
▶ group_2 = ["Fiona"]
```

```
[15] group
```

```
↳ ['An', 'Binh', 'Cuong', 'Dai', 'En']
```

Figure: Here we assign a new value to *group_2* using an assignment statement, *group_2* will point to a new object, while *group* stays the same.

Instantiation

- Instantiation is the process of creating a new object (instance) of a class
- For example `s1 = Student('An', 'VEF Academy', 'FML')` creates a student of VEF Academy's FML course named An
- Some built-in classes of Python also support the **literal** form of instantiation.
- E.g. The command `temperature=98.6` creates a new object of the **float** class
- The term 98.6 is a literal form
- In the canonical form, it may look something like
`temperature = float(98.6)`

Calling Methods

- In Python, we can call independent (non-class) functions that are invoked with a syntax such as `calculate_mean(my_list)`, in which case `my_list` is a parameter passed into the function.
- We can also define methods (member functions) for Python's classes, which can be invoked on an instance of the class using the dot (".") operator.
- Methods that return information about the state of an object are called **accessors**
- Methods that change the state of an object are called **mutators** or **update methods**

A Python class

```
class MyList():  
    def __init__(self, in_list):  
        # Data members (attributes)  
        self._list = in_list  
  
        # Member functions (methods)  
    def calculate_mean(self):  
        # Here we use the independent function  
        # calculate_mean() provided earlier  
        return calc_mean(self._list)  
  
scores = MyList([4, 5, 10])  
scores.calculate_mean()
```


Python's Built-In Classes

- The table below [1] provides a summary of commonly used, built-in classes in Python
- There are mutable and immutable classes.
- A class is immutable if each object of that class has a fixed value upon instantiation that cannot be changed later on.

| Class | Description | Immutable? |
|------------------|--------------------------------------|------------|
| bool | Boolean value | ✓ |
| int | integer (arbitrary magnitude) | ✓ |
| float | floating-point number | ✓ |
| list | mutable sequence of objects | |
| tuple | immutable sequence of objects | ✓ |
| str | character string | ✓ |
| set | unordered set of distinct objects | |
| frozenset | immutable form of set class | ✓ |
| dict | associative mapping (aka dictionary) | |

Table 1.2: Commonly used built-in classes for Python

- An **expression** consists of variables, values and **operators**.
- E.g., $a + b$ is an expression where a and b are variables and $+$ is an operator
- In **compound expressions** such as $a + b * c$, Python defines a specific order of precedence for evaluating operators.

Logical Operators

Python supports the following keyword operators for Boolean values:

| | |
|------------|-----------------|
| not | unary negation |
| and | conditional and |
| or | conditional or |

Equality Operators

Python supports the following operators to test two notions of equality:

| | |
|---------------|--------------------|
| is | same identity |
| is not | different identity |
| == | equivalent |
| != | not equivalent |

Comparison Operators

Data types may define a natural order via the following operators:

- < less than
- <= less than or equal to
- > greater than
- >= greater than or equal to

Arithmetic Operators

Python supports the following arithmetic operators:

| | |
|----|---------------------|
| + | addition |
| - | subtraction |
| * | multiplication |
| / | true division |
| // | integer division |
| % | the modulo operator |

Bitwise Operators

Python provides the following bitwise operators for integers:

- `~` bitwise complement (prefix unary operator)
- `&` bitwise and
- `|` bitwise or
- `^` bitwise exclusive-or
- `<<` shift bits left, filling in with zeros
- `>>` shift bits right, filling in with sign bit

Sequence Operators

| Python Data types | Description |
|--------------------|--------------------------------------|
| <code>list</code> | mutable sequence of objects |
| <code>tuple</code> | immutable sequence of objects |
| <code>str</code> | character string |

Sequence Operators

All sequences define comparison operations based on lexicographic order, performing an element by element comparison until the first difference is found. For example, $[5, 6, 9] < [5, 7]$ because of the entries at index 1. Therefore, the following operations are supported by sequence types:

- $s == t$ equivalent (element by element)
- $s != t$ not equivalent
- $s < t$ lexicographically less than
- $s <= t$ lexicographically less than or equal to
- $s > t$ lexicographically greater than
- $s >= t$ lexicographically greater than or equal to

Sequence Operators

| Sequence Operators | Description |
|---------------------------------|--|
| <code>s[i]</code> | Element at index <code>i</code> |
| <code>s[start:stop]</code> | Slice including indices <code>[start, stop)</code> |
| <code>s[start:stop:step]</code> | Slice including indices <code>start, start + step, start + 2*step,...</code> up to but not equaling or <code>stop</code> |
| <code>s + t</code> | Concatenation of sequences |
| <code>k * s</code> | Shorthand for <code>s + s + s + ...</code> (<code>k</code> times) |
| <code>val in s</code> | Containment check |
| <code>val not in s</code> | Non-containment check |

Operators for Sets and Dictionaries

| | | |
|---------------------------|---|-------|
| <code>key in s</code> | containment | check |
| <code>key not in s</code> | non-containment | check |
| <code>s == t</code> | s is equivalent to t | |
| <code>s != t</code> | s is not equivalent to t | |
| <code>s <= t</code> | s is a subset of t | |
| <code>s < t</code> | s is a proper subset of t | |
| <code>s >= t</code> | s is a superset of t | |
| <code>s > t</code> | s is a proper superset of t | |
| <code>s t</code> | the union of s and t | |
| <code>s & t</code> | the intersection of s and t | |
| <code>s - t</code> | the set elements in s but not in t | |
| <code>s ^ t</code> | the set of elements in precisely s or t | |

Dictionaries

| | |
|-----------------------------|---|
| <code>d[key]</code> | value associated with given key |
| <code>d[key] = value</code> | set (or reset) the value associated with given key |
| <code>del d[key]</code> | remove key and its associated value from dictionary |
| <code>key in d</code> | containment check |
| <code>key not in d</code> | non-containment check |
| <code>d1 == d2</code> | d1 is equivalent to d2 |
| <code>d1 != d2</code> | d1 is not equivalent to d2 |

Extended Assignment Operators

```
alpha = [1, 2, 3]
beta = alpha           # an alias for alpha
beta += [4, 5]         # extends the original list with two more elements
beta = beta + [6, 7]   # reassigns beta to a new list [1, 2, 3, 4, 5, 6, 7]
print(alpha)           # will be [1, 2, 3, 4, 5]
```

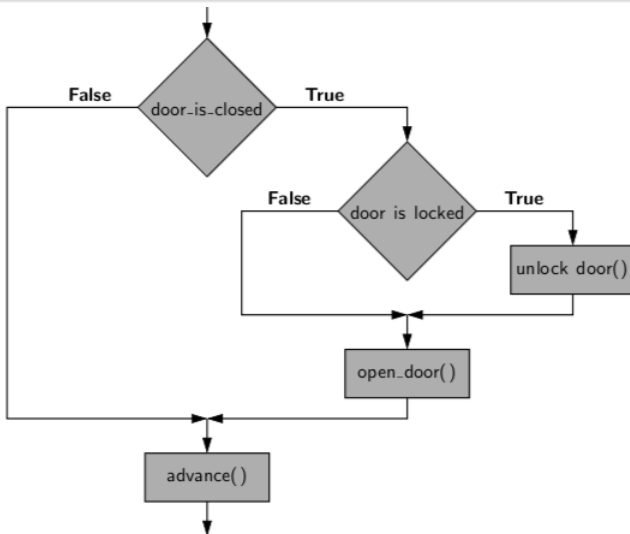
Compound Expressions and Operator Precedence

| Operator Precedence | | |
|---------------------|--|----------------------------------|
| | Type | Symbols |
| 1 | member access | expr.member |
| 2 | function/method calls container subscripts/slices | expr(...) expr[...] |
| 3 | exponentiation | ** |
| 4 | unary operators | +expr, -expr, ~expr |
| 5 | multiplication, division | *, /, //, % |
| 6 | addition, subtraction | +, - |
| 7 | bitwise shifting | <<, >> |
| 8 | bitwise-and | & |
| 9 | bitwise-xor | ^ |
| 10 | bitwise-or | |
| 11 | comparisons containment | is, is not, ==, !=, <, <=, >, >= |
| 12 | logical-not | not expr |
| 13 | logical-and | and |
| 14 | logical-or | or |
| 15 | conditional | val1 if cond else val2 |
| 16 | assignments | =, +=, -=, *=, etc. |

Conditionals

```
if first_condition:  
    first_body  
elif second_condition:  
    second_body  
elif third_condition:  
    third_body  
else:  
    fourth_body
```

An example flowchart



While loops

while condition:
 body

As an example, here is a loop that advances an index through a sequence of characters until finding an entry with value 'X' or reaching the end of the sequence.

```
j = 0
while j < len(data) and data[j] != 'X':
    j += 1
```

For loops

for element in iterable:

body # body may refer to element as an identifier

Index-Based For Loops

```
big_index = 0  
for j in range(len(data)):  
    if data[j] > data[big_index]:  
        big_index = j
```

Break and Continue Statements

```
found = False  
for item in data:  
    if item == target:  
        found = True  
        break
```

Break Statements

Listing 1: Break statement

```
found = False
for student_name in fml_class_list:
    if student_name == "Y Nguyen":
        found = true
        break
```

Continue Statements

Listing 2: Continue statement

```
count = 0
for student_name in fml_class_list:
    if student_name == "Y Nguyen":
        continue
    count += 1
```

Functions

```
def count(data, target):  
    n = 0  
    for item in data:  
        if item == target:           # found a match  
            n += 1  
    return n
```

Functions

- Information passing
- Mutable parameters
- Default parameter values
- Keyword parameters

Functions

Listing 3: Function example in Python

```
def count_students_by_name(  
    dsa_class_list,  
    name_of_interest  
):  
    count = 0  
    for student_name in dsa_class_list:  
        if student_name == name_of_interest:  
            count += 1  
    return count
```

High-level Languages

- Represent a giant leap towards easier programming
- Syntax similar to the English language
- Divided into 2 groups
 - Procedural languages
 - Object-oriented languages

Procedural Languages

- Specify the sequence of steps that implements a particular Algorithms
- Revolves around keeping code as concise as possible
- Focus on a very specific end result to be achieved
- Examples:
 - C
 - Fortran
 - Pascal

Object-Oriented Languages

- Focus not on structure, but on modeling data
- Programmers code using blueprints of data models called classes

Object-Oriented Programming

- A design program philosophy
- Key idea: the real world can be accurately described as a collection of objects that interact
- Everything is grouped as objects → enhance reusability

OOP Basic Terminology

- Object
- Method
- Attribute
- Class

Classes and Objects

- A class is a prototype, idea, and **blueprint** for creating objects
- An object is an **instance** of a class
- A class has a constructor for creating objects
- A class is composed of 3 things
 - Name
 - Attributes
 - Methods

Formal Definition of Objects

- A computational entity that:
 - Encapsulates some state
 - Is able to perform actions/methods on its state
 - Communicates with other objects

Classes and Objects Examples

Listing 4: Class example Python

```
class Human:
    def __init__(self, name):
        self.name = name

    def getName(self):
        return self.name

thai = Human("Thai Dinh")
```

OOP Basic Concepts

- Encapsulation
- Inheritance
- Abstraction

Encapsulation

- Inclusion of properties and methods within a class/object
- Enable reusability

Encapsulation

- Inclusion of properties and methods within a class/object
- Enable reusability

Inheritance

- Allow class hierarchy
- Enable reusability

Abstraction

- Allows programmers to represent complex real world in the simplest manner
- When we design abstract classes, we define the framework for later extensions

Advantages of OOP

- Code reuse and recycling
- Improved productivity
- Improved maintainability
- Faster development
- Higher quality, lower cost of software development

Disadvantages of OOP

- Steep learning curve
- Could lead to larger programs
- Could produce slower programs

References

- [1] M. H. Goldwasser, M. T. Goodrich, and R. Tamassia – Data Structures and Algorithms in Python, John Wiley & Sons, 2013
- [2] J. Unpingco, Python for Probability, Statistics, and Machine Learning (1st. ed.). Springer Publishing Company, Incorporated, 2016