Data Science and Machine Learning for Engineering Applications

Lecture Notes 1: Python Programming

March 8, 2023 - Politecnico di Torino

1 Python Introduction

1.1 Why Python?

There are several reasons why Python has become the most popular programming language, especially in the scientific research community. First of all, the **simplicity of its syntax** allows developers to write compact programs with fewer lines of code than the other programming languages. Python's syntax is similar to that of the English language, thus enabling it to write clean code that is easy to read, understand and debug. For instance, Python relies on indentation and whitespace to define the scope of loops, functions, and classes. Other programming languages often use curly brackets for this purpose.

Finally, there is an incredible Python community. Plenty of useful guides, documentation, and libraries exist for solving various problems and not always implementing everything from scratch.

1.2 Python Key Features

Python is an **interpreted** language. This means that code can be executed as soon as it is written. In contrast, many other programming languages are compiled and require an initial compilation phase that produces an executable that can then be executed (e.g., Java and C).

Python can be treated in a **procedural** way. You can code either in a **object-oriented** or a **functional** way. You can learn more about the differences between object-oriented and functional programming here.

2 Variables and simple Data Types

2.1 Variables naming

You should be aware of some rules and guidelines when naming variables to make code easier to read and understand and avoid errors [1].

- Variable names can only contain letters, numbers, and underscores. A variable name can start with a letter or an underscore but not a number.
- Variable names cannot contain whitespaces. To separate words in variables, you can use underscores.
- Avoid using Python keywords for function and variable names (i.e., print, max, etc.)
- Use descriptive but compact variable names.

2.2 Numbers

2.2.1 Integers

An **integer** is a number without decimal digits. You can add (+), subtract (-), multiply (*), divide (/), and exponentiate (**) integers in Python.

```
print(6 + 3)
print(6 - 3)
print(6 * 3)
print(6 / 3)
print(2 ** 3)
print((2 + 3) * 2)
print(2 + 3 * 2)
```

Output:

```
9
3
18
2
10
8
```

2.2.2 Floats

A **float** is a number with decimal digits. You can add (+), subtract (-), multiply (*), divide (/), and exponentiate (**) floats in Python with the same operands as for integers.

```
print(6 + 0.5)
print(6 - 3.5)
print(6 * 0.1)
print(2 / 0.5)
```

Output:

```
6.5
2.5
0.6
4.0
```

2.2.3 Casting

The conversion from one data type into another is called type **casting** or type conversion in python. Python supports a wide variety of functions or methods for type casting like `int()`, `float()`, `str()`, `tuple()`, `set()`, `list()`, `dict()`, etc.

```
print(int(3.14))
print(float(1))
```

Output:

```
3
1.0
```

2.3 Strings

A **string** is a sequence of characters and it is defined between quotes (single or double).

```
s1 = "this is a string"
s2 = "this is also a string"
print(type(s1))
print(type(s2))
```

Output:

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

There are many built-in methods for strings in Python. For instance, the `lower()` method puts all the string characters in lowercase. In contrast, the `upper()` method puts all the string characters in uppercase.

```
s1 = "This is a String"
print(s1.lower())
print(s1.upper())
```

Output:

```
this is a string"
"THIS IS A STRING"
```

2.3.1 String concatenation

String concatenation can be performed with the `+` operator.

```
s1 = "This is a first string"
s2 = ", this is a second string"
s3 = s1 + s2
print(s3)
```

Output:

```
"This is a first string, this is a second string"
```

3 Lists

A **list** is an **ordered** collection of items. They are defined between square brackets `[]`, and each individual element is separated by commas. Lists can contain any kind of item (e.g., strings, integers, floats, etc.). Lists are **mutable**.

```
my_list = ["apple", 2, "banana", 4]
```

3.1 Accessing list elements

You can access any element in a list by specifying the **index** (i.e., the position). You should write the index of the element in square brackets. Remember that in Python, the collection indices starts from 0.

```
my_list = ["apple", 2, "banana", 4]
print(my_list[0])
print(my_list[2])
```

Output:

```
"apple"
"banana"
```

3.2 Modifying list elements

To modify an element of a list, you should assign a new value to that element by accessing the index of the element that you want to modify.

```
my_list = ["apple", 2, "banana", 4]
print(my_list)
my_list[1] = 10
print(my_list)
```

Output:

```
["apple", 2, "banana", 4]
["apple", 10, "banana", 4]
```

3.3 Appending elements to the end of a list

You should use the `append()` method to add an element to the end of a list. The only parameter is the value that you want to insert.

```
my_list = ["apple", 2, "banana", 4]
print(my_list)
my_list.append("orange")
my_list.append("3")
print(my_list)
```

Output:

```
["apple", 2, "banana", 4]
["apple", 2, "banana", 4, "orange", 3]
```

This method makes it easy to build lists dynamically.

```
my_list = []
print(my_list)
my_list.append("apple")
my_list.append("2")
my_list.append("banana")
my_list.append("4")
print(my_list)
```

Output:

```
[]
["apple", 2, "banana", 4]
```

3.4 Inserting elements into a list

You should use the `insert()` method to add an element to a list. It takes two parameters: the index of the new element and the value.

```
my_list = ["apple", 2, "banana", 4]
print(my_list)
my_list.insert(0, "orange")
print(my_list)
```

Output:

```
["apple", 2, "banana", 4]
["orange", "apple", 2, "banana"]
```

3.5 Removing elements from a list

You have three options to remove elements from a list:

1. `Del` statement. You can remove the item in a particular index from a list using the `del` statement. However, after the removal, you can no longer access the value that was removed.

```
my_list = ["apple", 2, "banana", 4]
print(my_list)
del my_list[2]
print(my_list)
```

Output:

```
["apple", 2, "banana", 4]
["apple", 2, 4]
```

2. `Pop` method. The `pop()` method removes the item in the index provided as a parameter from a list and returns its value. In this case, you can access the value of the removed item. If you don't specify an index, the last item will be removed.

```
my_list = ["apple", 2, "banana", 4, "orange", 3]
print(my_list)
removed_item_1 = my_list.pop()
print(my_list)
removed_item_2 = my_list.pop(2)
print(my_list)
print("First item removed:", removed_item_1)
print("Second item removed:", removed_item_2)
```

`Output:`

```
["apple", 2, "banana", 4, "orange", 3]
["apple", 2, "banana", 4, "orange"]
["apple", 2, 4, "orange"]
"First item removed:  3"
"Second item removed:  banana"
```

3. `Remove` method. You can remove an item by value with the `remove()` method. It will remove only the **first occurrence** of the value specified as parameter.

```
my_list = ["apple", 2, "banana", 4, "banana", 2]
print(my_list)
my_list.remove("banana")
print(my_list)
my_list.remove(4)
print(my_list)
```

`Output:`

```
["apple", 2, "banana", 4, "banana", 2]
["apple", 2, 4, "banana", 2]
["apple", 2, "banana", 2]
```

3.6 Number of elements of a list

You can find the number of elements in a list using the `len()` function.

```
my_list = ["apple", 2, "banana", 4, "orange", 3]
print(len(my_list))
```

`Output:`

```
6
```

3.7 Sorting elements of a list

To sort the elements of a list, you can use either the `sort()` method or the `sorted()` function. The difference is that the `sort()` method is applied in place (i.e., orders the elements of the list and saves directly in the list itself). In contrast, the `sorted()` function returns a sorted list, but the values of the original list are unchanged.

```
my_list = [2, 1, 4, 5, 3]
print(my_list)
my_list.sort()
print(my_list)
```

Output:

```
[2, 1, 4, 5, 3]
[1, 2, 3, 4, 5]
```

```
my_list = [2, 1, 4, 5, 3]
print(my_list)
my_sorted_list = sorted(my_list)
print("my_sorted_list:", my_sorted_list)
print("my_list:", my_list)
```

Output:

```
[2, 1, 4, 5, 3]
my_sorted_list:  [1, 2, 3, 4, 5]
my_list:  [2, 1, 4, 5, 3]
```

3.8 Slicing a list

You can access sublists (as for substrings or tuples) by specifying the start (included) and stop (excluded) indices with the following syntax `[start:stop]`. You can omit the start or the stop indices to start from the beginning or go until the end, respectively. Remember that in python, the starting position is 0.

```
my_list = [1, 2, 3, 4, 5]
print(my_list)
print(my_list[:3]) # From the beginning until the fourth (excluded)
print(my_list[3:]) # From the fourth until the end
print(my_list[1:3]) # From the second until the fourth (excluded)
```

Output:

```
[1, 2, 3, 4, 5]
[1, 2, 3]
[4, 5]
[2, 3]
```

3.9 Statistics for list of numbers

You can also compute simple statistics to list of numbers using, for example, the `max()`, `min()`, and `sum()` functions:

```
my_list = [1, 2, 3, 4]
print(max(my_list))
print(min(my_list))
print(sum(my_list))
```

Output:

```
4
1
10
```

4 Loops

If you want to perform actions through all items in a sequence, you can use `for loops`. In this way, you can automate repetitive tasks. You should use the keyword `for` and indent the block that you want to reapeat with a \`tab`. This example iterates over the entire loop and prints the value of each item:

```
my_list = ["apple", 2, "banana", 4]
for el in my_list:
    print(el) #This is the code that will be repeated, indented with a \tab
```

Output:

```
"apple"
2
"banana"
4
```

This example iterates over the entire loop and prints the value of each item incremented by 1:

```
my_list = [1, 2, 3, 4]
for el in my_list:
    print(el + 1)
```

Output:

```
2
3
4
5
```

You can generate a series of numbers and iterate over it with the `range()` function. For example, if you want to print the numbers from 1 to n (both included):

```
n = 5
for i in range(1, n+1): # remember that sequences starts from 0
    print(i)
```

Output:

```
1
2
3
4
5
```

If you don't specify an initial value, the `range()` function starts from 0:

```
n = 5
for i in range(n+1): # remember that sequences starts from 0
    print(i)
```

Output:

```
0
1
2
3
4
5
```

If you want to iterate over a list, and perform some actions to it, you can loop over the indexes of the list. This example iterates over the entire list and add 5 to each element:

```
my_list = [1, 2, 3, 4]
print(my_list)
for i in range(len(my_list)): # the i variable will have values in range [0, 4[
    my_list[i] += 5 # this line incrementes the i-th element of the list by 5
print(my_list)
```

Output:

```
[1, 2, 3, 4]
[6, 7, 8, 9]
```

Sometimes, you may need to access both the **index** and the **value**, in this case, you can use the `enumerate()` function. This example iterates over the entire list and prints, at each iteration, the index, and the value:

```
my_list = ["orange", "apple", "banana", "avocado"]
for i, v in enumerate(my_list): # the i variable will have values in range [0, 4[, the v
    variable each value of each item
    print(f"Index: {i}, Value: {v}")
```

`Output:`

```
Index:  0, Value:  orange
Index:  1, Value:  apple
Index:  2, Value:  banana
Index:  3, Value:  avocado
```

5 List comprehensions

List comprehensions can be used to combine lists and loops and write compact code. This example shows the code to compute the square of each element of a list:

```
my_list = [1, 2, 3, 4]
print(my_list)

for i in range(len(my_list)):
    my_list[i] = my_list[i]**2
print(my_list)
```

`Output:`

```
[1, 2, 3, 4]
[1, 4, 9, 16]
```

This is the corresponding code with list comprehensions:

```
my_list = [1, 2, 3, 4]
print(my_list)
my_list = [value**2 for value in my_list]
print(my_list)
```

`Output:`

```
[1, 2, 3, 4]
[1, 4, 9, 16]
```

You can learn more on list comprehensions here. Even if it is not mandatory to write list comprehension, you can find it in many codes.

6 Conditional statements

To run code blocks based on some conditions, you can use the `if/else/elif` statements.

```
my_list = ["apple" , "banana" , "Apple" , "Banana" , "APPLE"]
for el in my_list:
    if el == "apple":   # First condition cheked. If True is run this code
        print ("I found an apple")
    elif el == "banana": # If the first condition is not True, the second condition is
    checked. If True is run this code
        print ("I found a banana" )
    else: # If both conditions are False
        print ("Not an apple or banana")
```

Output:

```
"I found an apple!"
"I found a banana!"
"Not an apple or banana"
"Not an apple or banana"
"Not an apple or banana"
```

String equality is case-sensitive. If you want to ignore the case while checking for equality or inequality, you should use the `lower()` function.

```
my_list = ["apple" , "banana" , "Apple" , "Banana" , "APPLE"]
for el in my_list :
    if el.lower() == "apple":   # the lower() method puts all characters in lowercase
        print ("I found an apple")
    elif el.lower() == "banana":  # the lower() method puts all characters in lowercase
        print ("I found a banana")
    else:
        print ("Not an apple or banana")
```

Output:

```
"I found an apple!"
"I found a banana!"
"I found an apple!"
"I found a banana!"
"I found an apple!"
```

6.1 Inequality

This example prints the indices of all elements different from `"apple"`. The inequality symbol is `!=`.

```
my_list = ["apple", "banana", "Apple", "Banana", "APPLE"]
for i in range(len(my_list)):
    if my_list[i].lower() != "apple": # The lower() method writes all characters in
    lowercase
        print(i)
```

Output:

```
1
3
```

6.2 Multiple conditions

You can specify multiple conditions using `and`, `or`, `not`.

```
my_list = ["apple", "banana", "orange", "banana", "apple"]
for i in range(len(my_list)):
    if my_list[i].lower() == "apple" or my_list[i].lower() == "orange":
        print("index {}: {}".format(i, my_list(i)))
```

Output:

```
"index 0:  apple"
"index 2:  orange"
"index 4:  apple"
```

```
my_list = [11, 20, 18, 30, 29, 50]
for i in range(len(my_list)):
    if my_list[i] >= 18 and my_list[i]<30:
        print("index {} in range [18, 30[".format(i))
```

Output:

```
"index 1 in range [18, 30[
"index 2 in range [18, 30[
"index 4 in range [18, 30[
```

6.3 Check if a list contains an item

You can check if the list contains a particular item with the `in` operand.

```
my_list = ["apple", "banana", "orange", "banana", "apple"]
if "apple" in my_list:
    print("apple in the list")
else:
    print("apple not in the list")
```

Output:

```
"apple in the list"
```

If you want to check if a value is **not** present in the list, you can negate the `in` operand.

```
my_list = ["apple", "banana", "orange", "banana", "apple"]
if "car" not in my_list:
    print("car in the list")
else:
    print("car not in the list")
```

Output:

```
"car not in the list"
```

7 Dictionaries

A dictionary is a collection of **key-value** pairs. Each key is connected to a value, and you can use a key to easily access the value associated with that key. Every key is connected to its value by a colon, and individual key-value pairs are separated by commas. Keys must be unique. Values can be numbers, strings, lists, or other dictionaries. Dictionaries are **mutable**.

7.1 Check if a dictionary contains a key

To check if a dictionary contains a particular key you can use the `in` operator.

```
my_dict = {"apples":2 , "bananas":4, "oranges":6}
if "apple" in my_dict:
    print("apple in the dictionary")
else:
    print("apple not in the dictionary")
```

Output:

```
"apple in the dictionary"
```

7.2 Modify a dictionary value

If you want to modify the value of a key-value pair in a dictionary, you can assign the new value and access the relative key-value pair by key (with square brackets).

```
my_dict = {"apples":2, "bananas":4, "oranges":6}
print("before modification:", my_dict)
my_dict["apples"] = 10 # assign a new value
print("after modification:", my_dict)
```

Output:

```
before modification:  {"apples":2, "bananas":4, "oranges":6}
after modification:  {"apples":10, "bananas":4, "oranges":6}
```

7.3 Add a new key-value pair

You can add a new key-value pair in the same way (if the key is not present):

```
my_dict = {"apples":2, "bananas":4, "oranges":6}
my_dict["avocados"] = 10
print(my_dict)
```

Output:

```
{"apples":2, "bananas":4, "oranges":6, "avocados":10}
```

7.4 Iterate over the keys

To get all the keys in your dictionary, you can use the `keys()` method. It returns a view of all the keys in the dictionary.

```
my_dict = {"apples":2, "bananas":4, "oranges":6}
for k in my_dict.keys():
    print(f"key: {k}")
```

Output:

```
key:  apples
key:  bananas
key:  oranges
```

7.5 Iterate over the values

To get all the values in your dictionary, you can use the `values()` method. It returns a view of all the values in the dictionary.

```
my_dict = {"apples":2, "bananas":4, "oranges":6}
for v in my_dict.values():
    print(f"value: {v}")
```

Output:

```
value:  2
value:  4
value:  6
```

7.6 Iterate over the keys and values

You can use the `items()` method to get all the key-value pairs in your dictionary. It returns a view of all the key-value pairs in the dictionary.

```
my_dict = {"apples":2, "bananas":4, "oranges":6}
for k, v in my_dict.items():
    print(f"key: {k}, value: {v}")
```

Output:

```
key:  apples, value:  2
key:  bananas, value:  4
key:  oranges, value:  6
```

8 Booleans

A Boolean expression is just another name for a conditional test. A Boolean value is either `True` or `False`. As in one previous example, you want to check if the list contains a particular item with the `in` operand. However, you want to create a boolean with the value `True` is the item is in the list, `False` otherwise.

```
my_list = ["apple", "banana", "orange", "banana", "apple"]
if "apple" in my_list:
    my_flag = True
else:
    my_flag = False
print(my_flag)
```

Output:

```
True
```

9 Functions

Functions are useful to avoid repetitive code.

9.1 Function definition and invocation (call)

A function is defined with the `def` keyword, followed by the name of the function, the parameters between round brackets, and a column.

```

def my_print_fn(x): # function definition
    print(x)
    return          # return statement

my_print_fn("hello")  # function call
my_print_fn(10)
```

Output:

```
hello
10
```

9.2 Returning multiple values

A function can return multiple values. In this example, the function takes two parameters in input and returns their sum and difference.

```

def my_print_fn(x, y): # function definition
    my_sum = x + y
    my_diff = x - y
    return my_sum, my_diff          # return statement

my_sum, my_diff = my_print_fn(10, 5)  # function call
print(my_sum)
print(my_diff)
```

Output:

```
15
5
```

10 Accumulator pattern

Often, you need to scan a sequence variable (e.g., dicts, lists, etc.) and iteratively create a new list or compute an aggregated value of the elements that pass some conditions. In this example, we have a dictionary, and we want to: i) sum all the values of the key-value pairs that have a positive value; and ii) produce a list with all the keys of those key-value pairs. This can be solved with the accumulator pattern.

```
my_dict = {"oranges":-1, "apples":4, "bananas":-1, "avocados":6}
sum_positive = 0 # initialize the sum to 0
positive_keys = [] # initialize an empty list
for k, v in my_dict.items(): # scan all the key-value pairs
    if v >= 0: # check the wanted condition
        sum_positive += v # if the condition is true, perform the cumulative sum
        positive_keys.append(k) # and add the key

print(sum_positive)
print(positive_keys)
```

Output:

```
10
['apples', 'avocados']
```

References

[1] E. Matthes. *Python Crash Course, 2nd Edition: A Hands-On, Project-Based Introduction to Programming*. No Starch Press, 2019. ISBN: 9781593279288. URL: https://books.google.it/books?id=nkh8tgEACAAJ.