PySimp

Zafer Kosar

2024-06-09

Table of contents

Preface

This is a Quarto book.

To learn more about Quarto books visit https://quarto.org/docs/books.

Part I

Installation

1 Installing python

It is suggested use panultimate (one before the latest) major release of Python. Python 3 is now the golden standard and major releases follow 3.x.x convetion. So, if the latest release is **3.12.x**, use the **3.11.y**, y being the latest version, the reason being compatibility with 3rd party libraries takes some time. For this book, we will use Python 3.11.9.

1.1 Windows

On windows just go to https://www.python.org/downloads/ and scroll down to "Python releases by version number:" section and select a version to download. And continue with the installation.

On the installation, click "**Add python.exe to PATH**" so you can use Python on your terminal.

1.2 Mac

On Mac just go to https://www.python.org/downloads/ and scroll down to "Python releases by version number:" section and select a version to download. And continue with the installation.

2 Installing Third Party Libraries

Effective use of Python requires many third party libraries (packages). Most prominently `numpy` is used for working with arrays, `pandas` or `polars` for tabular data, `scipy` contains scientific algorithms etc., These third party libraries make Python very attractive as they solve the **speed problem** of Python while utilizing the ease-of-use of Python.

2.1 pip

To install third party libraries, we can use `pip` installer that comes built-in with Python installation. We add **`install` flag** for installation to declare our purpose. Lastly, `install` flag requires a parameter which is the library name e.g., `numpy`. Shortly, we can open a **terminal window** and write the following.

```
pip install numpy
```

This will directly install `numpy` from PyPi, the public repository of Python, to your Python packages.

! Note That in some cases `pip3` is used instead of `pip`

2.1.1 Installing Libraries on Notebook

In some cases you may want to or have to install libraries inside of a Jupyter Notebook or Notebook in general. In such cases, just add **exclammation mark** "!" or **percentage sign** "%" before `pip`.

```
%pip install numpy
```

2.1.2 Installing without admin priviliges

If you're working on an HPC or something similar you may not have admin priviliges. Therefore, you may not have to access to main folder of Python packages. As a work-around you can add `--user` flag to install the packages to your own packages.

```
pip install numpy --user
```

Above Python 3.10 installation defaults to user, so you don't need the `--user` flag.

Part II

Basics

3 Variables

A variable is used as shortcut for a **value**. Python has some fundamental variable types to hold distinct type of values.

The most notable of these fundamental variable types are;

3.1 Strings

Strings ,that are denoted by type `str` in python, used to hold *string* of characters.

```python
my_name = "Zafer Kosar the First of his name"
```

3.2 Floats and Integers

Numeric variables `float` for floating point numbers (decimal numbers) and `int` for integers (whole numbers).

```python
number_of_my_friends = 12 # I can't have a 0.35 friend.
type(number_of_my_friends) # this will output the type of the variable
```

```
int
```

```python
my_height_in_meters = 1.84 # Some variables require floating point numbers
type(my_height_in_meters)
```

```
float
```

3.3 Boolean

Logic based variables holds `True` or `False` **boolean** values denoted by `bool`. They can be used to assess equality of values and variables. They are very useful to check whether the **conditions** are satisfied, thus control the flow of the code.

As single equal sign `=` is reserved for assignment of value to variable. Double equal `==` sign is used for comparison.

An example comparison that will output `False`;

```
1 == 0
```

```
False
```

Also, variables can be used for comparison;

```
my_height_in_meters == my_name
```

```
False
```

As for the `type` of this assessment;

```
type(my_height_in_meters == my_name)
```

```
bool
```

An example that will output `True`;

```
2 == 2.0 # ! Important Note here: Unlike some other languages, in Python float 2 (2.0) is equ
```

```
True
```

As expected

```
type(2) == type(2.0) # int vs float
```

```
False
```

3.4 None

`None`,as the name suggests, is used for denoting that there is no value assigned to the variable. Note that it is different from not being defined at all.

We haven't talked about the functions yet. But, let's declare a simple function named `simple_function()` which returns **nothing**.

```
def simple_function():
    return
```

```
x = simple_function()
type(x)
```

```
NoneType
```

We can also declare a variable with a value of `None` to make it `NoneType`.

```
y = None
type(y)
```

```
NoneType
```

3.5 About variables

Variable names **must not** start with a number and **must not** include special characters except for underscore "_". Python uses *snake case* convention which states variable names (and each word in a variable name) **should** start with a lowercase letter and each word should be seperated by an underscore '_'.

Python is a **dynamically typed** language. So, variables types will be inferred by the interpreter. Although this makes Python a slower compared to **statically typed** languages (e.g., C, Rust, Java etc.,), it makes Python beginner friendly also reduces the development time.

```
num_samples : int = 33 # static typing
num_samples = "33" # dynamic typing
```

Although Python allows static typing (aka type hinting), it is **not required** and interpreter **ignores** it. However, they may be useful for the programmer to keep track of their variables.

4 Lists

Lists are used to store mutable **sequence** or **collection** of values (or items).

A list can be created with collection of values seperated by commas `,` and enclosed within `[]`

```
ids = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
ids
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

4.1 List methods

Items also can be appended to the **end** of the list with `append()` method of list object.

```
ids.append(11)
ids
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

or they can also be appended to the beginning or any given (existing) index of a list via `insert()` method.

```
ids.insert(0, -1) # at index 0, insert -1
ids
```

```
[-1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

It is possible to reverse a list via `reverse()` method.

```
ids.reverse()
ids
```

```
[11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, -1]
```

Any value in the list can also be removed via `remove()`. Note that, `remove()` only removes the first occurence of a value.

```
ids.remove(8)
ids
```

```
[11, 10, 9, 7, 6, 5, 4, 3, 2, 1, 0, -1]
```

You can get the methods you can call upon a **list** using the `dir()` function

```
dir(list)[-10:] # [-10:] gives the last 10 elements of the list returned by dir()
```

```
['clear',
 'copy',
 'count',
 'extend',
 'index',
 'insert',
 'pop',
 'remove',
 'reverse',
 'sort']
```

4.2 Built-in Functions to use with lists

Python has built-in `min()` and `max()` functions.

```
max(ids)
```

```
11
```

```
min(ids)
```

```
-1
```

Summation of all elements in the list are also possible.

```
sum(ids)
```

```
57
```

and the number of elements (length) in a list can be accessed via `len()` function.

```
len(ids)
```

```
12
```

Note that, `min()`, `max()`, and `len()` are not `methods` for `list` **Class** and they can be used with other collections of items such as **tuples** and **arrays**.

Lists objects have also be sorted via `sort()` method.

```
random_numbers = [5353, 314235, 353, 9,-12]
random_numbers.sort()
random_numbers
```

```
[-12, 9, 353, 5353, 314235]
```

Lists don't have to be collection of the same type values. **Any** Python object can be stored in a list including other lists.

```
mixed = ["a", "b", "c", 1, 10 , 100, True, False, ids, None, 3.14, random_numbers, "d", "e",
mixed
```

```
['a',
 'b',
 'c',
 1,
 10,
 100,
 True,
 False,
 [11, 10, 9, 7, 6, 5, 4, 3, 2, 1, 0, -1],
 None,
 3.14,
 [-12, 9, 353, 5353, 314235],
 'd',
 'e',
 'f']
```

Obiviously, you can not call `sort()`, `min()`, `max()` or similar methods on a mixed list.

4.3 Indexing

Like most programming languages index start from `0` in Python. And the index should put within `[]` next to the list name.

```
mixed[0] # this will return the first element of a list
```

```
'a'
```

To access the last element in a list. You can use `-1` as the index. Or `-2`, `-3` so on as a way of starting from the end.

```
mixed[-1]
```

```
'f'
```

is equivalent to

```
mixed[len(mixed) - 1]
```

```
'f'
```

4.4 Slicing

Let's say you want to get the first element on a list. You can use the `:5` (equivalent of `0:5`) for indexing that will return the first five elements.

```
mixed[:5]
```

```
['a', 'b', 'c', 1, 10]
```

Or you can get next 3 elements

```
mixed[5:8] # will return the elements from index 5 to index 8, where 8 is not included.
```

```
[100, True, False]
```

Or the last 5 elements

```
mixed[-5:]
```

```
[3.14, [-12, 9, 353, 5353, 314235], 'd', 'e', 'f']
```

Or you might want the every second element. You can use the `::2` which means from **beginning to end** (`::`) every 2nd (2).

```
mixed[::2]
```

```
['a', 'c', 10, True, [11, 10, 9, 7, 6, 5, 4, 3, 2, 1, 0, -1], 3.14, 'd', 'f']
```

Of course 2 can be changed with any number of your choosing.

```
mixed[::5]
```

```
['a', 100, 3.14]
```

Or you can get them in reverse order, basically reversing the list `::-1`

```
mixed[::-1]
```

```
['f',
 'e',
 'd',
 [-12, 9, 353, 5353, 314235],
 3.14,
 None,
 [11, 10, 9, 7, 6, 5, 4, 3, 2, 1, 0, -1],
 False,
 True,
 100,
 10,
 1,
 'c',
 'b',
 'a']
```

5 Tuples

5.1 todo

6 Dictionaries

6.1 TODO

Part III

Conditionals

7 Operators

Conditionals is used for controlling the flow of code. Operators are useful to check if an **input** or a **result** satisfies a certain condition. Your code can change its course or behavior depending on the conditions. These operations will return a `bool` type value.

```python
big = 5
mid = 4
small = 3
```

7.1 Greater than and Less than

Many of the operators in Python is similar to the mathematical notation.

The operators `>` (greater than) and `<` (less than) are exactly the same.

```python
big > small
```

```
True
```

```python
big < mid
```

```
False
```

7.2 Equality

Equality checks requires double equal sign `==`.

```python
big == mid
```

```
False
```

```
small == small
```

```
True
```

There is also **not equal** to operator `!=`

```
big != mid
```

```
True
```

7.3 Not

`not` keyword/operator can be used to invert `True` to `False` and vice versa.

```
not True
```

```
False
```

7.4 Chanining operations

Operators can also be chained via `and` and `or`.

`and` requires all of the conditions to be satisfied.

```
(5*1 == 15/3) and (5*1 == 10) # second condition is not satisfied so this will reutrn False
```

```
False
```

With `or` it is sufficient that one of conditions to be `True` to return `True`.

```
(5*1 == 15/3) or (5*1 == 10) # even though second condition is not satisfied, this will retu
```

```
True
```

Of course, you can use many `or` and `and` and even parantheses `()` to create much more complicated logical comparisons. Order of priority is from left to right with parantheses first manner.

8 If conditionals

8.1 If

`if` keyword in Python let the interpreter **run** a piece of code **codeblock** when a requirement satisfied. This lets the program control the workflow depending on the previous results or inputs.

```
input = 12 # just for simplicity
input2 = 17 # just for simplicity again
```

we will compare these input with the expected input.

an `if` block starts with an `if` followed by a comparison. This first line must end with an `:`. the next line and any line within the `if` block should have a *tab* (by convention 4 spaces) distance from the line start.

```
if input == 12:
    print("Hello World!")
```

```
Hello World!
```

```
if input2 == 12: # input2 is not 12, so this will not print anything
    print("Hello World!")
```

8.2 Elif

When we want to test for another set of conditions, we can use `elif` (short for else if).

```
if input2 == 12:
    print("Hello World!")
elif input2 == 13:
    print("Hello World but 13!")
elif input2 == 15:
    print("Hello World but 15!")
elif input2 == 17: # only this will print
    print("Hello World but 17!")
```

```
Hello World but 17!
```

8.3 Else

When you want to execute a piece of code when no other condition is satisfied you can simply use `else`.

```
if input2 == 12:
    print("Hello World!")
elif input2 == 13:
    print("Hello World but 13!")
else: # only this will print
    print("Hello World but no prior conditions satisfied!")
```

```
Hello World but no prior conditions satisfied!
```

8.4 Chaining Conditions

It is possible to chain conditions with `and` and `or`.

```
if input == 12 and input2 == 17: # both conditions must be true
    print("Hello World! Both conditions satisfied!")
```

```
Hello World! Both conditions satisfied!
```

Another example with `or`

```
if input == 17 or input2 == 17: # only second condition is True
    print("Hello World! One or both conditions satisfied!")
```

```
Hello World! One or both conditions satisfied!
```

8.5 Nested Conditions

An if-else code block can be inside another if-else block which can be inside another if-else block and so on. This modularity enables programmer to program complex workflow with their codes.

First, let's define an Arthur person for a follow up nested condition demonstration.

```
age_arthur = 23
height_aurthur = 187
weight_arthur = 88
name_arthur = "Arthur"
gender_arthur = "male"
hobby_arthur = None
royal_blood_index_arthur = 0.7
is_arthur_king = False
```

```
is_arthur_king
```

```
False
```

Now, let's see if we can crown him as the king...

```
# if is_arthur_king == True is the same as below since is_arthur_king is already a boolean
if is_arthur_king:
    print("Arthur is already a king! Cannot crown him again!")
else:
    if gender_arthur !="male": # if he is not male
        print("Cannot be a king without being male")
    else: # if he is male
        if age_arthur < 18: # check if he is old enough
            print("Cannot be a king without being old enough")
        else: # if he is old enough
            if royal_blood_index_arthur < 0.5: # check if he has enough royal blood
                print("Cannot be a king without enough royal blood")
            else: # if he has enough royal blood
                if height_aurthur < 180 or weight_arthur < 80: # check if he is big enough
                    print("Cannot be a king without being huge enough")
                else: # if he is tall and heavy enough
                    if hobby_arthur is not None:
                        print("Cannot be a king with a hobby, Kings don't have time for hobb
                    else:
                        print("Long live the King Arthur!")
                        is_arthur_king = True # now he is a king and we can update his statu
```

```
Long live the King Arthur!
```

Now, let's see if Arthur is a King now!

```
is_arthur_king
```

```
True
```

We used nested (one within another) if-else blocks to check multiple scenarios. This is of course a made up example and can be simplified with `and` and `or` operators. But for real life cases you may encounter a problem in which you will have to use nested conditions to control flow of logic.

With this we arrive to the conclusion we should crown him King and executed the command via the assigment `is_arthur_king = True` This assigment here is a simple command. But we could have much more complex operations or combinations of operations to execute.

Part IV

Loops

9 For Loops

Loops are used to iterate over a collection of elements such as list, arrays, tuples etc., or to iterate over an **iterator** object.

9.1 Iterators

Maybe the most used iterator in Python is from `range()` function. This function will return an iterator that begins at zero (or at the given number) up to **end** number inclusively. Unlike lists they store values one-by-one rather than all at the same time.

To first visualize what it produces we can convert a `range` object into a `list`.

```
list(range(1, 11)) #start=1 and stop=11, start is included but stop is not included.
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

if you only input one value, that value will be treated as stop value. Start is by default `0` and this is more useful than starting from `1` as indexes start from `0`.

```
list(range(8))
```

```
[0, 1, 2, 3, 4, 5, 6, 7]
```

Increment value is by default `1` but can be changed by adding a third parameter

```
list(range(0, 100, 10))
```

```
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90]
```

9.2 Looping over iterators

Similar to `if` blocks, `for` loop block starts with `for` followed by an `in` **iterable** and ends with `:`. Following lines within the for block must start with a **tab**.

```python
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

10