Introduction to Python Day 3

Verjinia Metodieva and Daniel Parthier

2025-02-18

Index based *for loops* - range()

- generates integer sequences
- range(n) generates the series of n values from 0 to n − 1

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

```
0
1
2
3
4
```

```
# looping through data indices. find the max
B = [1, 4, 6, 7, 89, 54]
big_indx = 0
for i in range(len(B)):
    if B[i] > B[big_indx]:
        big_indx = i
print('The max value in B is', B[big_indx], 'found on position', big_indx)
```

```
The max value in B is 89 found on position 4
```

Index based *for loops* - enumerate()

- assigns a count to each item within an iterable and returns it as an enumerate object
- one way to avoid nested loops

```
import numpy as np

array_a = np.arange(20, 25)
for indx, val in enumerate(array_a):
    print('the index is', indx)
    print('the value is', val)
```

```
the index is 0
the value is 20
the index is 1
the value is 21
the index is 2
the value is 22
the index is 3
the value is 23
the index is 4
the value is 24
```

! range() and enumerate() - none of the two returns a list of objects!

- motivation: limitation in 'simple' for loops - we don't know the position of an element within a sequence, as we experienced in the last example
- range(n) generates the series of n values from 0 to n − 1
- precisely the series of valid indices into a sequence of length n
- range() - returns a range object that is iterable
- enumerate() - returns an enumerate object that is also iterable
- they are mainly used in loops

Break and continue statements

- break - immediately terminates the loop
- continue - skips whatever is after it and continues with the next iteration
  - mostly used after a conditional statement

- a break statement that immediately terminates a while or for loop when executed within its body
- a continue statement - skips the rest of the statements in the current iteration of the loop and it returns control to the beginning of the loop

While loops

- Perform a task `while` something is `True`
- Be careful:
  - Some loops never finish (get stuck)
  - Make sure that condition for ending the loop can be fullfilled

```
while check_condition:
    perform_task()
```

If your python terminal gets stuck at one point you can try a `KeyboardInterrupt` using `Ctrl+C`, which will kill the running script.

Let's wait while we wait

- Start a little counter

```
import time
counter = 0
while counter < 10:
    time.sleep(1)
    counter += 1
    print("You waited for " + str(counter) + " seconds...")
```

- Good for keeping processes running

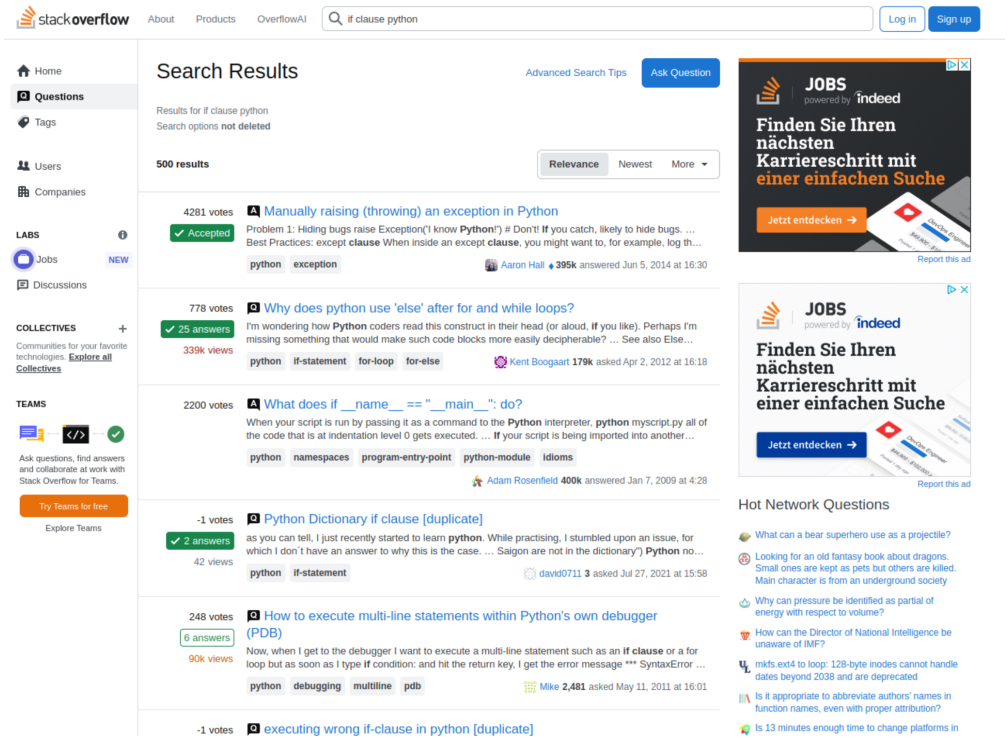
Try to avoid `while` loops as much as possible. They can be useful if you do not have information how long it should run, but know it will at one point finish.

Errors and how to read them

There are useful resources regarding errors

- Simply googling works surprisingly well
- You will often end up on stackoverflow
  - There is no question which was not already asked[1]

[1] if that is not true open up a question



Types of errors

1. SyntaxErrors
2. IndentationError
3. NameError
4. TypeError
5. IndexError
6. AttributeError
7. etc.

Fix errors

- Breath
- Don't panic
  - Identify the error by checking the terminal output

– Look at the line provided
– Go backwards if error is nested