הטכניון
היחידה ללימודי המשך
ולימודי חוץ עייש עזריאלי

Builtins and One Liners

TECHNION
Azrieli Continuing Education and
External Studies Division

Builtins

Our goal is to minimize the amount of time it takes to write code.

We can do this using the many features of Python.

If we learn more features and shortcuts, our code will be shorter, easier to read, and more efficient.

Min / Max

The **`min()`** and **`max()`** functions receive a sequence and return the minimum or maximum values.

```
In [1]: lst = [0, 1, 1, 4, 6, -4, 2]

In [2]: min(lst)
Out[2]: -4

In [3]: max(lst)
Out[3]: 6
```

Sum

The **sum()** function receives a sequence of numbers and returns their sum.

```
In [6]: sum(range(4))
Out[6]: 6
```

for Loop with Multiple Variables

We can use a **for** loop with multiple variables, which will be unpacked.

For this to work, you need to give the loop a sequence, in which every item is a tuple or a list that has the same amount of inner items.

```
In [7]: for i, j in [(1, 11), (2, 22), (3, 33)]:
   ...:     print(i, j, j - i)
   ...:
1 11 10
2 22 20
3 33 30
```

enumerate

If we want to iterate over a sequence:

```
In [22]: for i in 'abc':
    ...:     print(i)
    ...:
a
b
c
```

enumerate

Now, we also want to print the index of the item. We will have to use range, with indexing:

```
In [23]: seq = 'abc'

In [24]: for i in range(len(seq)):
    ...:     print(i, seq[i])
    ...:
0 a
1 b
2 c
```

enumerate

This is a lot easier done with the **`enumerate()`** function.

```
In [25]: seq = 'abc'

In [26]: for i, ltr in enumerate(seq):
    ...:     print(i, ltr)
    ...:
0 a
1 b
2 c
```

Assigning a Variable with a Condition

We can also make a conditional variable assignment like this:

```
In [34]: str_num = '17.5'

In [35]: if '.' in str_num:
    ...:     num = float(str_num)
    ...: else:
    ...:     num = int(str_num)
    ...:
```

a lot shorter, with a one-liner:

```
In [32]: str_num = '17.5'

In [33]: num = float(str_num) if '.' in str_num else int(str_num)
```

List Comprehension

List Comprehension is one of the strongest and **best** features in Python!

The syntax is easy: Inside square brackets, we write the expression we want to every item to be, and then a regular “for i in sequence”.

```
In [36]: [i for i in range(4)]
Out[36]: [0, 1, 2, 3]

In [37]: [i + 1 for i in range(4)]
Out[37]: [1, 2, 3, 4]

In [38]: [i ** 2 for i in range(4)]
Out[38]: [0, 1, 4, 9]
```

List Comprehension – cont.

```
In [39]: words = ['yes', 'inshallah', 'happy']

In [40]: [len(word) for word in words]
Out[40]: [3, 9, 5]
```

We can even add an “if” to it!

```
In [41]: days = ['Sun', 'Mon', 'Tues', 'Wed', 'Thur', 'Fri', 'Sat']

In [42]: [day for day in days if day.startswith('T')]
Out[42]: ['Tues', 'Thur']
```

List Comprehension – Pros and Cons

<u>Pros:</u>

- Quick and easy to write
- Makes your code shorter and easier to read
- More efficient than "**append**"ing to a list

<u>Cons:</u>

- There aren't any.
  - Seriously, list comprehension is awesome.

Dict Comprehension

Very similar to List Comprehension, but this time it creates a dict.

We use curly (instead of square) brackets, and for every item we write a *key: value* pair.

```
In [44]: {i: str(i) for i in range(4)}
Out[44]: {0: '0', 1: '1', 2: '2', 3: '3'}

In [45]: {i: i ** 2 for i in range(4)}
Out[45]: {0: 0, 1: 1, 2: 4, 3: 9}
```

Lambda

Lambda functions are functions you can write in one line.

They should be very quick to write, and shouldn't be important, complicated, or reused a lot.

After the keyword lambda, we write our arguments (divided by commas, then a colon, then the output of the function.

```
In [46]: square = lambda x: x ** 2

In [47]: square(4)
Out[47]: 16

In [48]: type(square)
Out[48]: function
```

```
In [49]: is_even = lambda x: x % 2 == 0

In [50]: is_even(7)
Out[50]: False
```

Lambda – Pros and Cons

Pros:

- Quick to write when we need a simple function

Cons:

- No documentation / docstring
- Hard to debug, with no way to print values in the middle

Map

The **`map()`** function receives a function and a sequence.

It then runs the function on every item in the sequence, returning a sequence of the results.

```
In [56]: list(map(str, range(5)))
Out[56]: ['0', '1', '2', '3', '4']

In [57]: list(map(type, [1, 1.0, True, 'hey', None]))
Out[57]: [int, float, bool, str, NoneType]
```

Map – cont.

```
In [58]: list_of_lists = [[0, 0], [0], [0, 0, 0, 0]]

In [59]: list(map(len, list_of_lists))
Out[59]: [2, 1, 4]
```

It is very useful to use **map** with **lambda**!

```
In [61]: list(map(lambda x: x ** 2, range(1, 6)))
Out[61]: [1, 4, 9, 16, 25]
```

Filter

The **`filter()`** function also receives a function and a sequence.

The function should return True/False.

Filter runs the function on each item in the sequence, and returns a new sequence of all the items that returned True.

```
In [62]: lst = [1, 0, 0.0, [], 'a', '']

In [63]: list(filter(bool, lst))
Out[63]: [1, 'a']
```

Filter and lambda

```
In [64]: list(filter(lambda x: x % 2 == 0, range(10)))
Out[64]: [0, 2, 4, 6, 8]
```

```
In [66]: nums = [1, -1, 2, 4, 6, -7, -10, 0]

In [67]: list(filter(lambda x: x > 0, nums))
Out[67]: [1, 2, 4, 6]
```

Summary

- Min, Max, Sum
- For loop with unpacking
- Enumerate
- If-else one-liner
- List Comprehension
- Dict Comprehension
- Lambda functions
- Map
- Filter