

Week 9 Tutorial: Fundamentals of Python Programming II

POP77001 Computer Programming for Social Scientists

Indentation

- Use tabs (1 tab) or spaces (4) consistently in your code
- Python also permits mixing of different indentation levels, avoid it for legibility 📄

```
1 # Only for illustration purposes, do not do in practice!  
2 # 2 spaces before print() statement  
3 for i in range(5):  
4     print(i, end = ' ')
```

0 1 2 3 4

```
1 # 4 spaces before print() statement  
2 for i in range(5):  
3     print(i, end = ' ')
```

0 1 2 3 4

Indentation and Readability

- Main rule, be consistent!
- Think not just whether Python throws an error, but also readability

```
1 # This is semantically valid, but is badly styled
2 l = [0, 1, 1, 5]
3 for i in l:
4     if i % 2 == 1: # 2 spaces
5         print(i) # 6 spaces
6 print('End')
```

```
1
1
5
End
```

Iterables

- An object is an iterable if it has `__iter__` method that can be called with `iter()` function

```
1 x = 3
2 iter(x)
```

TypeError

Traceback (most recent call last)

Cell In[48], line 2

```
1 x = 3
----> 2 iter(x)
```

TypeError: 'int' object is not iterable

```
1 y = 'abc'
2 iter(y)
```

<str_ascii_iterator at 0x764074257550>

Iteration over Dictionaries

- `items()` method allows to iterate over keys and values in a dictionary
- `keys()` method allows to iterate over just keys
- `values()` method allow to iterate over just values

```
1 d = {'apple': 150.0, 'banana': 120.0, 'watermelon': 3000.0}
```

```
1 for k, v in d.items():  
2     print(k.upper(), int(v))
```

```
APPLE 150  
BANANA 120  
WATERMELON 3000
```

```
1 for k in d.keys():  
2     print(k.title())
```

```
Apple  
Banana  
Watermelon
```

```
1 for v in d.values():  
2     print(str(v/1000) + ' kg')
```

```
0.15 kg  
0.12 kg  
3.0 kg
```

List Comprehensions

- The same iteration can often be implemented with **for** loop block or list comprehension
- The choice is often between less typing, speed (🚗) and legibility (📄)

```
[<expr> for <elem> in <iterable>]  
[<expr> for <elem> in <iterable> if <test>]  
[<expr> for <elem1> in <iterable1> for <elem2> in <iterable2>]
```

E.g. the following conditional statement:

```
1 l = [0, 'one', 1, 2]  
2 for i in range(len(l)):  
3     if isinstance(x, int):  
4         l[i] = x * 2
```

Can be implemented as a list comprehension:

```
1 l = [0, 'one', 1, 2]  
2 [x * 2 for x in l if isinstance(x, int)]
```

```
[0, 2, 4]
```

Exercise: List comprehensions and **for** loops

- Consider a list of International vehicle registration codes below.
- Suppose we want to create a list where each element is the length of each string in this list.
- First, implement it using a **for** loop.
- Now try doing the same using a list comprehension.
- Finally, modify the list comprehension to keep only those elements that start with D.
- You can use string method **startswith** for the last task.

```
1 l = ['D', 'DK', 'EST', 'F', 'IRL', 'MD', 'NL', 'S', 'UK']
```


Set and Dictionary Comprehensions

- Analogous to list, sets and dictionaries have their own concise ways of iterating over them.
- Note that iterating over them tends to be slower than over lists (.

```
{<expr> for <elem> in <iterable> if <test>}  
{<key>: <value> for <elem1>, <elem2> in <iterable> if <test>}
```

```
1 o = {'apple', 'banana', 'watermelon'}  
2 {e[0].title() + ' - ' + e for e in o}
```

```
{'A - apple', 'B - banana', 'W - watermelon'}
```

```
1 d = {'apple': 150.0, 'banana': 120.0, 'watermelon': 3000.0}  
2 {k.upper(): int(v) for k, v in d.items()}
```

```
{'APPLE': 150, 'BANANA': 120, 'WATERMELON': 3000}
```

Note of Caution

- Avoid modifying a sequence that you are iterating over.
- As this can lead to unexpected results:

```
1 l = [x for x in range(1,11)]  
2 l
```

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

```
1 for i in l:  
2     print('Element - ' + str(i))  
3     if i % 2 == 0:  
4         l.pop(i)  
5     print('Length = ' + str(len(l)))
```

Element - 1

Length = 10

Element - 2

Length = 9

Element - 4

Length = 8

Element - 5

```
Length = 8  
Element - 7  
Length = 8  
Element - 8
```

IndexError

Traceback (most recent call last)

Cell In[60], line 4

```
    2 print('Element - ' + str(i))  
    3 if i % 2 == 0:  
----> 4     l.pop(i)  
    5 print('Length = ' + str(len(l)))
```

IndexError: pop index out of range

Docstring

- **Docstring** provides a standardized way of documenting functionality.
- It is defined as a first statement in module, function, class, or method definition.
- Docstring is accessible with `help()` function.
- It also creates a special `__doc__` attribute.



Python documentation on docstring

One-line Docstrings

- In the simplest case, docstrings can take just one line:

```
def <function_name>(arg_1, arg_2, ..., arg_n):  
    """<docstring>"""  
    <function_body>
```

```
1 def add_one(x):  
2     """Adds 1 to numeric input"""  
3     return x + 1
```

```
1 help(add_one)
```

```
Help on function add_one in module __main__:
```

```
add_one(x)  
    Adds 1 to numeric input
```

```
1 add_one.__doc__
```

```
'Adds 1 to numeric input'
```

Multi-line Docstrings

- A more elaborate docstring would consist of a single summary line, followed by a blank line, followed by a longer description of inputs, arguments and output:

```
def <function_name>(arg_1, arg_2, ..., arg_n):  
    """<summary_docstring>
```

```
  
    <longer_description>  
    """
```

```
    <function_body>
```

```
1  def even_or_odd(num):  
2      """Check whether the number is even or odd  
3  
4      Takes an integer as input  
5      Returns the result as a string  
6      """  
7      if num % 2 == 0:  
8          return 'even'  
9      else:  
10         return 'odd'
```

```
1  help(even_or_odd)
```

Help on function even_or_odd in module __main__:

```
even_or_odd(num)  
    Check whether the number is even or odd  
  
    Takes an integer as input  
    Returns the result as a string
```

Exercise: Functions and Docstrings

- Most functions for calculating summary statistics would be available in separate packages (built-in `statistics` and external `numpy`).
- But it is helpful to try programming some of those yourself to understand the internal working.
- Modify the function definition below according to its docstring specification.
- You can use function `round` for rounding.
- Try your function with `0.1`, `2.7`, `3.5`, `4`, `5.98` supplied as arguments.

```
1 def calculate_mean():
2     """
3     Calculates mean
4
5     Takes any number of numeric arguments as an input.
6     Returns mean rounded to two decimal place.
7     """
8     pass
```

Week 9: Assignment 3

- Python Fundamentals and Control Flow
- Due by 12:00 on Monday, 11th November