Week 9 Tutorial: Fundamentals of Python Programming II

POP77001 Computer Programming for Social Scientists

Module website: tinyurl.com/POP77001

Indentation

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

Indentation

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

In [1]:

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

```
0 1 2 3 4
```

Indentation

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

In [1]:

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

```
0 1 2 3 4
```

In [2]:

```
# 4 spaces before print() statement
for i in range(5):
    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

Indentation and readability

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

In [3]:

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

```
1
1
5
End
```

Check whether object is iterable

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

Check whether object is iterable

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

In [4]:

```
x = 3
iter(x)
```

```
---------------------------------------------------------------
-------------
TypeError                                 Traceback (most recen
t call last)
Input In [4], in <cell line: 2>()
      1 x = 3
----> 2 iter(x)

TypeError: 'int' object is not iterable
```

Check whether object is iterable

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

In [4]:
```
x = 3
iter(x)
```

```
---------------------------------------------------------------
------------
TypeError                                 Traceback (most recen
t call last)
Input In [4], in <cell line: 2>()
      1 x = 3
----> 2 iter(x)

TypeError: 'int' object is not iterable
```

In [5]:
```
y = 'abc'
iter(y)
```

Out[5]:
```
<str_iterator at 0x7ff44c13fb50>
```

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

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

In [6]:

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

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

In [6]:
```
d = {'apple': 150.0, 'banana': 120.0, 'watermelon': 3000.0}
```

In [7]:
```
for k, v in d.items():
    print(k.upper(), int(v))
```

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

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

In [6]:

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

In [7]:

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

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

In [8]:

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

```
Apple
Banana
Watermelon
```

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

In [6]:
```
d = {'apple': 150.0, 'banana': 120.0, 'watermelon': 3000.0}
```

In [7]:
```
for k, v in d.items():
    print(k.upper(), int(v))
```

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

In [8]:
```
for k in d.keys():
    print(k.title())
```

```
Apple
Banana
Watermelon
```

In [9]:
```
for v in d.values():
    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>]
```

Exercise 1: 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.

Exercise 1: 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.

In [10]:

```
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>}
```

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>}
```

```
In [14]: o = {'apple', 'banana', 'watermelon'}
         {e[0].title() + ' - ' + e for e in o}

Out[14]: {'A - apple', 'B - banana', 'W - watermelon'}
```

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>}
```

In [14]:

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

Out[14]:

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

In [15]:

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

Out[15]:

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

Note of caution

- Avoid modifying a sequence that you are iterating over
- This can lead to unexpected results

Note of caution

- Avoid modifying a sequence that you are iterating over
- This can lead to unexpected results

In [16]:

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

Out[16]:

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

Note of caution

- Avoid modifying a sequence that you are iterating over
- This can lead to unexpected results

```
In [16]:
l = [x for x in range(1,11)]
l
```

```
Out[16]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
In [17]:
for i in l:
    print('Element - ' + str(i))
    if i % 2 == 0:
        l.pop(i)
    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)
Input In [17], in <cell line: 1>()
      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

Extra: Python documentation on docstring

One-line docstrings

- In the simplest case, docstrings can take only one line

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

One-line docstrings

- In the simplest case, docstrings can take only one line

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

In [18]:

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

One-line docstrings

- In the simplest case, docstrings can take only one line

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

In [18]:

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

In [19]:

```python
help(add_one)
```

```
Help on function add_one in module __main__:

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

One-line docstrings

- In the simplest case, docstrings can take only one line

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

```
In [18]: def add_one(x):
             """Adds 1 to numeric input"""
             return x + 1
```

```
In [19]: help(add_one)
```

```
Help on function add_one in module __main__:

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

```
In [20]: add_one.__doc__
```

```
Out[20]: '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>
```

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>
```

In [21]:

```python
def even_or_odd(num):
    """Check whether the number is even or odd

    Takes an integer as input
    Returns the result as a string
    """
    if num % 2 == 0:
        return 'even'
    else:
        return 'odd'
```

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>
```

In [21]:

```
def even_or_odd(num):
    """Check whether the number is even or odd

    Takes an integer as input
    Returns the result as a string
    """
    if num % 2 == 0:
        return 'even'
    else:
        return 'odd'
```

In [22]:

```
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 2: Functions

- 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.

Exercise 2: Functions

- 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.

In [23]:

```python
def calculate_mean():
    """
    Calculates mean

    Takes any number of numeric arguments as an input.
    Returns mean rounded to two decimal place.
    """
    pass
```

Week 9: Assignment 3

- Python Fundamentals and Control Flow
- Due by 12:00 on Monday, 14th November