

Lecture 1.2: Python Basics

March 31, 2022

Python Files

- You can write and edit code in files. This is the preferred method when you're working on a large codebase or repeatedly editing code.
- Code that should only be executed when the file is being called directly is placed in:

```
if __name__ == '__main__':  
    # only executes if this file is being called directly  
    ...
```

- Execute the file by calling `python file.py`

Python Style



(a stylish python)

Comments

Comments

```
# A single-line comment in Python is denoted with the hash symbol.
```

Comments

```
# A single-line comment in Python is denoted with the hash symbol.
```

```
"""  
Multi-line comments  
Lie between quotation marks  
This is a haiku  
"""
```

PEP 8

PEP 8

Spacing

PEP 8

Spacing

Use four spaces to indent code (don't use tabs).

PEP 8

Spacing

Use four spaces to indent code (don't use tabs).

Use blank lines to separate functions from each other and logical sections within a function.

PEP 8

Spacing

Use four spaces to indent code (don't use tabs).

Use blank lines to separate functions from each other and logical sections within a function.

Use spaces around operators and after commas, but not directly inside delimiters.

```
a = f(1, 2) + g(3, 4) # good
a = f( 1, 2 ) + g( 3, 4 ) # bad
```

PEP 8

PEP 8

Commenting

PEP 8

Commenting

Comment all nontrivial functions.

A function's docstring is the *first (unassigned) string* inside the function body.

PEP 8

Commenting

Comment all nontrivial functions.

A function's docstring is the *first (unassigned) string* inside the function body.

Describe parameters (value / expected type) and return (value / expected type).

PEP 8

Commenting

Comment all nontrivial functions.

A function's docstring is the *first (unassigned) string* inside the function body.

Describe parameters (value / expected type) and return (value / expected type).

As usual: list pre/post conditions if any.

PEP 8

Commenting

Comment all nontrivial functions.

A function's docstring is the *first (unassigned) string* inside the function body.

Describe parameters (value / expected type) and return (value / expected type).

As usual: list pre/post conditions if any.

Add header comments at the top of files before any imports.

PEP 8

Commenting

Comment all nontrivial functions.

A function's docstring is the *first (unassigned) string* inside the function body.

Describe parameters (value / expected type) and return (value / expected type).

As usual: list pre/post conditions if any.

Add header comments at the top of files before any imports.

If possible, put comments on a line of their own.


```
def my_function():  
    """  
    Summary line: do nothing, but document it.  
  
    Longer description: No, really, it doesn't do anything.  
  
    Returns: Gosh, for the last time... nothing (None)!  
    """  
    pass
```

```
def my_function():  
    """  
    Summary line: do nothing, but document it.  
  
    Longer description: No, really, it doesn't do anything.  
  
    Returns: Gosh, for the last time... nothing (None)!  
    """  
    pass
```

```
print(my_function.__doc__)  
#     Summary line: do nothing, but document it.  
#  
#     Longer description: No, really, it doesn't do anything.  
#  
#     Returns: Gosh, for the last time... nothing (None)!
```

PEP 8

PEP 8

Naming

PEP 8

Naming

Use `snake_case` for variables/functions; `CamelCase` for classes;
`CAPS_CASE` for constants.

PEP 8

Naming

Use `snake_case` for variables/functions; `CamelCase` for classes;
`CAPS_CASE` for constants.

Decomposition and Logic

PEP 8

Naming

Use `snake_case` for variables/functions; `CamelCase` for classes; `CAPS_CASE` for constants.

Decomposition and Logic

Same as 106A/B/X. Simple is better than complex!

PEP 8

Naming

Use `snake_case` for variables/functions; `CamelCase` for classes; `CAPS_CASE` for constants.

Decomposition and Logic

Same as 106A/B/X. Simple is better than complex!

Automated Code Style Checking

PEP 8

Naming

Use `snake_case` for variables/functions; `CamelCase` for classes; `CAPS_CASE` for constants.

Decomposition and Logic

Same as 106A/B/X. Simple is better than complex!

Automated Code Style Checking

Use [PEP8 Online](#) for mechanical violations (naming, spacing) and more advanced suggestions.

PEP 8

Naming

Use `snake_case` for variables/functions; `CamelCase` for classes; `CAPS_CASE` for constants.

Decomposition and Logic

Same as 106A/B/X. Simple is better than complex!

Automated Code Style Checking

Use [PEP8 Online](#) for mechanical violations (naming, spacing) and more advanced suggestions.

Use `pycodestyle` as a command line tool. Install with `pip install pycodestyle` (you'll do this in the installation instructions).

Review Activity: Seesaw

- Introduce yourself to your neighbor — well-being inquiries are in order!
- Work through this problem together:

<https://edstem.org/us/courses/20141/lessons/32533>

File I/O

```
f = open(filename, method)
```

Read

Function	Action
<code>next(f)</code>	Returns the next line in the file
<code>f.read()</code>	Returns the entire file as a string
<code>for line in f:</code>	Loops over the file, line by line
<code>f.readlines()</code>	Returns the lines of the file as a list of strings

Write

Function	Action
<code>f.write(new_line)</code>	Writes <code>new_line</code> to the file
<code>f.writelines([collection of new, lines])</code>	Writes the collection of lines to the file

** Writing appends or overwrites, depending on the method*

```
f.close()
```



```
f = open(filename, method)
```

- r - Read
- w - Write
- a - Append
- b - Bytes Mode

Read

Function	Action
<code>next(f)</code>	Returns the next line in the file
<code>f.read()</code>	Returns the entire file as a string
<code>for line in f:</code>	Loops over the file, line by line
<code>f.readlines()</code>	Returns the lines of the file as a list of strings

Write

Function	Action
<code>f.write(new_line)</code>	Writes <code>new_line</code> to the file
<code>f.writelines([collection of new, lines])</code>	Writes the collection of lines to the file

** Writing appends or overwrites, depending on the method*

```
f.close()
```

What happens without `f.close()`?

What happens without `f.close()`?

- When the program ends (naturally or from an error), Python will try to clean up any objects that remain in memory.

What happens without `f.close()`?

- When the program ends (naturally or from an error), Python will try to clean up any objects that remain in memory.
- This isn't guaranteed*, but it happens sometimes. You should be concerned if you're writing code that will be run on many operating systems or Python versions.

* Depends on the operating system, Python version, Python implementation (which language the interpreter was written in), ...

What happens without `f.close()`?

- When the program ends (naturally or from an error), Python will try to clean up any objects that remain in memory.
- This isn't guaranteed*, but it happens sometimes. You should be concerned if you're writing code that will be run on many operating systems or Python versions.
- If it isn't closed, the file could remain locked so other programs can't open it or become corrupted.

* Depends on the operating system, Python version, Python implementation (which language the interpreter was written in), ...

What happens without `f.close()`?

- When the program ends (naturally or from an error), Python will try to clean up any objects that remain in memory.
- This isn't guaranteed*, but it happens sometimes. You should be concerned if you're writing code that will be run on many operating systems or Python versions.
- If it isn't closed, the file could remain locked so other programs can't open it or become corrupted.
- The safe option: use a **context manager**!

* Depends on the operating system, Python version, Python implementation (which language the interpreter was written in), ...

```
with open("words.txt", "r") as f:
```

```
with open("words.txt", "r") as f:
```

`open("words.txt", "r")` is a file object - it has instructions about how to open and close the file.

The context manager makes sure those instructions are followed, no matter what.


```
with open("words.txt", "r") as f:
```

`open("words.txt", "r")` is a file object - it has instructions about how to open and close the file.

The context manager makes sure those instructions are followed, no matter what.



```
with open("words.txt", "r") as f:
```

`open("words.txt", "r")` is a file object - it has instructions about how to open and close the file.

The context manager makes sure those instructions are followed, no matter what.

Roughly equivalent to:

```
f = open("words.txt", "r")  
try:  
    ...  
finally:  
    f.close()
```



Strings, Revisited

Useful String Methods

Method	Action
<code>.lower()</code>	Converts the string to lowercase
<code>.upper()</code>	Converts the string to uppercase
<code>.title()</code>	Converts the string to title case (every word capitalized)
<code>.strip([chars])</code>	Removes the characters from the ends of the string (or whitespace if <code>chars</code> is omitted)

Method	Action
<code>.find(substr)</code>	Finds the first occurrence of <code>substr</code> and returns the index (or -1 if not found)
<code>.replace(old, new)</code>	Replaces every instance of <code>old</code> with <code>new</code> and returns the new string
<code>.startswith(substr)</code> <code>.endswith(substr)</code>	Returns whether the string starts/ends with <code>substr</code>

Useful String Methods

Method	Action
<code>.lower()</code>	Converts the string to lowercase
<code>.upper()</code>	Converts the string to uppercase
<code>.title()</code>	Converts the string to title case (every word capitalized)
<code>.strip([chars])</code>	Removes the characters from the ends of the string (or whitespace if <code>chars</code> is omitted)

Method	Action
<code>.find(substr)</code>	Finds the first occurrence of <code>substr</code> and returns the index (or -1 if not found)
<code>.replace(old, new)</code>	Replaces every instance of <code>old</code> with <code>new</code> and returns the new string
<code>.startswith(substr)</code> <code>.endswith(substr)</code>	Returns whether the string starts/ends with <code>substr</code>

Splitting and Joining

```
"3-14-2015".split('-')
```

Splitting and Joining

```
"3-14-2015".split('-')
```

Splitting and Joining

```
"3-14-2015".split('-') # => ['3', '14', '2015']
```


Splitting and Joining

```
"3-14-2015".split('-') # => ['3', '14', '2015']
```

```
"Tara Elizabeth Jones".split()
```

Splitting and Joining

```
"3-14-2015".split('-') # => ['3', '14', '2015']
```

```
"Tara Elizabeth Jones".split()
```

Splitting and Joining

```
"3-14-2015".split('-') # => ['3', '14', '2015']
```

```
"Tara Elizabeth Jones".split()  
# => ['Tara', 'Elizabeth', 'Jones']
```

Splitting and Joining

```
"3-14-2015".split('-') # => ['3', '14', '2015']
```

```
"Tara Elizabeth Jones".split()  
# => ['Tara', 'Elizabeth', 'Jones']
```

```
", ".join(["Minerva", "Albus", "Severus"])
```

Splitting and Joining

```
"3-14-2015".split('-') # => ['3', '14', '2015']
```

```
"Tara Elizabeth Jones".split()  
# => ['Tara', 'Elizabeth', 'Jones']
```

```
", ".join(["Minerva", "Albus", "Severus"])  
# => 'Minerva, Albus, Severus'
```