# Lecture 1.2: Python Basics

*April 5, 2023*

# Learning Goals

After today, students will be able to…

- Build a chatbot in Python.
- Define Python terms like “module” and “workspace” and create new modules on their computer.
- Identify resources for determining whether they are using appropriate Python style.

# Making a module:

`is_prime.py`

# 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.

```python
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
```

```python
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

For the remaining concepts:

`chatbot.py`

# File I/O

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

*Read*

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

*Write*

| Function | Action |
|---|---|
| `f.write(new_line)` | Writes `new_line` to the file |
| `f.writelines([collection, of, new, lines])` | 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 |
|---|---|
| `next(f)` | Returns the next line in the file |
| `f.read()` | Returns the entire file as a string |
| `for line in f:` | Loops over the file, line by line |
| `f.readlines()` | Returns the lines of the file as a list of strings |

*Write*

| Function | Action |
|---|---|
| `f.write(new_line)` | Writes `new_line` to the file |
| `f.writelines([collection, of, new, lines])` | Writes the collection of lines to the file |

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

```
f.close()
```

Add a file read loop:

`chatbot.py`

# 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()
```

Safely read definitions:

`chatbot.py`

# Strings, Revisited

# Useful String Methods

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

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

# Useful String Methods

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

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

# 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(["Arpit", "Chase", "Will"])
```

# Splitting and Joining

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

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

", ".join(["Arpit", "Chase", "Will"])
# => 'Arpit, Chase, Will'
```

Finish the chatbot:

`chatbot.py`

# Intro to Data Structures

# First, a summary

| | mutable? | ordered? | iterable? | check inclusion | delimiters |
|---|---|---|---|---|---|
| list | ✔ | ✔ | over the entries | `O(n)` | `[ ]` |
| tuple | ✘ | ✔ | over the entries | `O(n)` | `( )` |
| set | ✔ | ✘ | over the entries | `O(1)` | `{ }` |
| dictionary | ✔ | ✘ | over the keys | `O(1)` for the keys | `{ }` |

# Lists

```
to_remember = ['car keys', 'grading', 'the alamo', 42]
```

**Lists are...**

- **mutable** — they can be changed after they're created

```
to_remember.remove(42)          # O(n)
to_remember.append('september') # O(1)
```

- **ordered** — there's a 0th element, 1st element, 2nd element, ...

```
to_remember[3] # => 'september'
```

- **heterogeneous** — they can store elements of different types

# Lists

# Lists

| | |
|---|---|
| `.count(elem)` | **Counts the occurrences of elem in the list.** |

# Lists

| | |
|---|---|
| **`.count(elem)`** | **Counts the occurrences of elem in the list.** |
| `.index(elem)` | Returns the index of the first occurrence of elem in the list. |

# Lists

| | |
|---|---|
| **`.count(elem)`** | **Counts the occurrences of elem in the list.** |
| `.index(elem)` | Returns the index of the first occurrence of elem in the list. |
| **`.append(elem)`** | **Appends the element elem to the end of the list.** |

# Lists

| | |
|---|---|
| **`.count(elem)`** | **Counts the occurrences of elem in the list.** |
| `.index(elem)` | Returns the index of the first occurrence of elem in the list. |
| **`.append(elem)`** | **Appends the element elem to the end of the list.** |
| `.extend(iterable)` | Extends the list by appending all elements of iterable to the end. |

# Lists

| | |
|---|---|
| **`.count(elem)`** | **Counts the occurrences of elem in the list.** |
| `.index(elem)` | Returns the index of the first occurrence of elem in the list. |
| **`.append(elem)`** | **Appends the element elem to the end of the list.** |
| `.extend(iterable)` | Extends the list by appending all elements of iterable to the end. |
| `.insert(idx, elem)` | Inserts the element elem at the index idx of the list. |

# Lists

| | |
|---|---|
| **`.count(elem)`** | **Counts the occurrences of elem in the list.** |
| `.index(elem)` | Returns the index of the first occurrence of elem in the list. |
| **`.append(elem)`** | **Appends the element elem to the end of the list.** |
| `.extend(iterable)` | Extends the list by appending all elements of iterable to the end. |
| `.insert(idx, elem)` | Inserts the element elem at the index idx of the list. |
| `.sort(key=None, reverse=False)` | Sorts the list in-place. |

# Lists

| | |
|---|---|
| **`.count(elem)`** | **Counts the occurrences of elem in the list.** |
| `.index(elem)` | Returns the index of the first occurrence of elem in the list. |
| **`.append(elem)`** | **Appends the element elem to the end of the list.** |
| `.extend(iterable)` | Extends the list by appending all elements of iterable to the end. |
| `.insert(idx, elem)` | Inserts the element elem at the index idx of the list. |
| `.sort(key=None, reverse=False)` | Sorts the list in-place. |
| **`elem in let`** | **Returns True if elem is in the list and False otherwise** |

# Lists

| | |
|---|---|
| **`.count(elem)`** | **Counts the occurrences of elem in the list.** |
| `.index(elem)` | Returns the index of the first occurrence of elem in the list. |
| **`.append(elem)`** | **Appends the element elem to the end of the list.** |
| `.extend(iterable)` | Extends the list by appending all elements of iterable to the end. |
| `.insert(idx, elem)` | Inserts the element elem at the index idx of the list. |
| `.sort(key=None, reverse=False)` | Sorts the list in-place. |
| **`elem in let`** | **Returns True if elem is in the list and False otherwise** |
| `del lst[i]` | Removes the ith element from the list |

# Lists

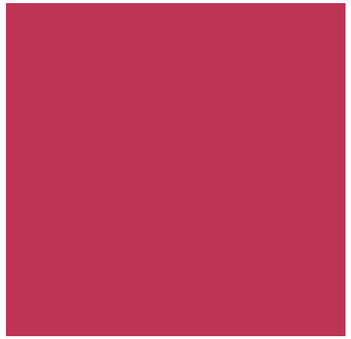
| | |
|---|---|
| **`.count(elem)`** | **Counts the occurrences of elem in the list.** |
| `.index(elem)` | Returns the index of the first occurrence of elem in the list. |
| **`.append(elem)`** | **Appends the element elem to the end of the list.** |
| `.extend(iterable)` | Extends the list by appending all elements of iterable to the end. |
| `.insert(idx, elem)` | Inserts the element elem at the index idx of the list. |
| `.sort(key=None, reverse=False)` | Sorts the list in-place. |
| **`elem in let`** | **Returns True if elem is in the list and False otherwise** |
| `del lst[i]` | Removes the ith element from the list |
| **`.pop(i=-1)`** | **Returns and removes the ith element from the list.** |

# Lists

| | |
|---|---|
| **`.count(elem)`** | **Counts the occurrences of elem in the list.** |
| `.index(elem)` | Returns the index of the first occurrence of elem in the list. |
| **`.append(elem)`** | **Appends the element elem to the end of the list.** |
| `.extend(iterable)` | Extends the list by appending all elements of iterable to the end. |
| `.insert(idx, elem)` | Inserts the element elem at the index idx of the list. |
| `.sort(key=None, reverse=False)` | Sorts the list in-place. |
| **`elem in let`** | **Returns True if elem is in the list and False otherwise** |
| `del lst[i]` | Removes the ith element from the list |
| **`.pop(i=-1)`** | **Returns and removes the ith element from the list.** |
| **`.remove(elem)`** | **Removes the first instance of elem from the list, or raises ValueError.** |

# Tuples

```
pix = (190, 52, 85)
```




**Tuples are...**

- **immutable** — can't be changed after creation (consequently, they're hashable)

```
pix[2] = 210 # TypeError: 'tuple' does not support assignment
hash(pix)    # => 8626792735414146673
```

- **ordered** — there's a 0th element, 1st element, 2nd element, ...

```
pix[0] # => 190
```

- **heterogeneous** — they can store elements of different types

# Tuples

**Tuples are...**

- **immutable** — can't be changed after creation (consequently, they're hashable)

**Immutability is powerful!**

- When you guarantee that you're not going to change the entries, they can be stored in a slightly more efficient way
- Tuples can be hashed if they contain immutable data structures — remember this for later!
- Tuples contain immutable *references*...

```
tup = (1, 2, [3, 4])
tup[2].append(5)
tup # => (1, 2, [3, 4, 5])
```

This is totally valid, but inadvisable!

# Putting it together: `filter_pixels`

filter_pixels.py

```
def is_bright(r, g, b):
    avg_val = (r + g + b) / 3
    return avg_val >= 128

def filter_pixels(pixels):
    # apply is_bright to filter the list
    ...

filter_pixels([
  (11, 231, 128), (224, 178, 46), (226, 226, 133), (225, 83, 205),
  (37, 89, 102), (119, 67, 141), (170, 239, 125), (135, 22, 2),
  (83, 105, 96), (16, 19, 96)
])
```

# Sets

```
tas = {'chase', 'arpit', 'will', 'chase', 41}
```

**Sets are...**

- **mutable** — they can be changed after they're created

```
tas.add('arpit') # O(n)
tas.remove(41)   # O(1)
```

- **unordered** — there's no guarantee which element you'll pop

```
tas.pop() # => 'will'
```

- **heterogeneous** — they can store elements of different types
- **unique** — they remove duplicates; every element of a set must be hashable (for now, just think each element must be immutable)

```
tas # => {'chase', 'arpit'}
```

# Sets

**Sets are... mathematical objects!**

| | |
|---|---|
| **`s & t`** | **Set intersection.** |
| **`s \| t`** | **Set union.** |
| `s < t` | Check whether s is a proper subset of t. |
| **`s <= t`** | **Check whether s is a subset of t.** |
| `s ^ t` | Symmetric difference. |
| **`s - t`** | **Set difference.** |

# Mathematical sets and efficient phrases

| These are efficient phrases | These aren't efficient phrases |
|---|---|
| COLD WINDOWSILL | CHILLY WINDOW LEDGE |
| COOL MILLION | GOOD THOUSAND THOUSAND |
| VIVID DISILLUSIONS | GRAPHIC DISAPPOINTMENTS |
| SUSPICIOUS CONCLUSION | MISTRUSTFUL ENDING |

What makes an efficient phrase?

# Dictionaries

```
passwords = {
    'tara': 'ilovecs41',
    'arpit': None,
    'chase': 'pyth0nrock$'
}
```

**Dictionaries are...**

- **mutable** — they can be changed after they're created

```
pswds['arpit'] = 'un1c0rn$4lyfe'
del pswds['chase']
```

- **associative** — access values by keys, not position (no 0th, 1st, 2nd, ... element)
- **heterogeneous** — they can store elements of different types
- **unique keys** — each key can only appear once, keys must be hashable

# Dictionaries

# Dictionaries

| | |
|---|---|
| `val = d[key]` | Access the value in d corresponding to key; place this value into the val variable. |

# Dictionaries

| | |
|---|---|
| `val = d[key]` | Access the value in d corresponding to key; place this value into the val variable. |
| `d.copy()` | Makes a shallow copy of d. |

# Dictionaries

| | |
|---|---|
| `val = d[key]` | Access the value in d corresponding to key; place this value into the val variable. |
| `d.copy()` | Makes a shallow copy of d. |
| `d.get(key, default)` | Returns the value associated with key in d. If key does not exist in d, return default. |

# Dictionaries

| | |
|---|---|
| `val = d[key]` | Access the value in d corresponding to key; place this value into the val variable. |
| `d.copy()` | Makes a shallow copy of d. |
| `d.get(key, default)` | Returns the value associated with key in d. If key does not exist in d, return default. |
| `d.keys()` | Returns a collection of the keys in the dictionary. |

# Dictionaries

| | |
|---|---|
| `val = d[key]` | Access the value in d corresponding to key; place this value into the val variable. |
| `d.copy()` | Makes a shallow copy of d. |
| `d.get(key, default)` | Returns the value associated with key in d. If key does not exist in d, return default. |
| `d.keys()` | Returns a collection of the keys in the dictionary. |
| `d.values()` | Returns a collection of the values in the dictionary. |

# Dictionaries

| | |
|---|---|
| `val = d[key]` | Access the value in d corresponding to key; place this value into the val variable. |
| `d.copy()` | Makes a shallow copy of d. |
| `d.get(key, default)` | Returns the value associated with key in d. If key does not exist in d, return default. |
| `d.keys()` | Returns a collection of the keys in the dictionary. |
| `d.values()` | Returns a collection of the values in the dictionary. |
| `d.items()` | Returns a collection of (key, value) tuples in d. |

# Dictionaries

| | |
|---|---|
| `val = d[key]` | Access the value in d corresponding to key; place this value into the val variable. |
| `d.copy()` | Makes a shallow copy of d. |
| `d.get(key, default)` | Returns the value associated with key in d. If key does not exist in d, return default. |
| `d.keys()` | Returns a collection of the keys in the dictionary. |
| `d.values()` | Returns a collection of the values in the dictionary. |
| `d.items()` | Returns a collection of (key, value) tuples in d. |
| `d.clear()` | Removes all (key, value) pairs from d. |

# Dictionaries

| | |
|---|---|
| `val = d[key]` | Access the value in d corresponding to key; place this value into the val variable. |
| `d.copy()` | Makes a shallow copy of d. |
| `d.get(key, default)` | Returns the value associated with key in d. If key does not exist in d, return default. |
| `d.keys()` | Returns a collection of the keys in the dictionary. |
| `d.values()` | Returns a collection of the values in the dictionary. |
| `d.items()` | Returns a collection of (key, value) tuples in d. |
| `d.clear()` | Removes all (key, value) pairs from d. |
| `d.pop(key, default)` | Removes key, and its associated value, from d. (Returns the associated value if key is in d, otherwise returns default). |

# First, a summary

| | mutable? | ordered? | iterable? | check inclusion | delimiters |
|---|---|---|---|---|---|
| list | ✔ | ✔ | over the entries | `O(n)` | `[ ]` |
| tuple | ✘ | ✔ | over the entries | `O(n)` | `( )` |
| set | ✔ | ✘ | over the entries | `O(1)` | `{ }` |
| dictionary | ✔ | ✘ | over the keys | `O(1)` for the keys | `{ }` |

# Patterns for working with collections

# Patterns for working with collections

# Patterns for working with collections

```
# number of elements in a collection
```

# Patterns for working with collections

```
# number of elements in a collection
len(collection)
```

# Patterns for working with collections

```
# number of elements in a collection
len(collection)
```

# Patterns for working with collections

```
# number of elements in a collection
len(collection)

# loop over the elements in a collection
```

# Patterns for working with collections

```
# number of elements in a collection
len(collection)

# loop over the elements in a collection
for elem in collection:
```

# Patterns for working with collections

```
# number of elements in a collection
len(collection)

# loop over the elements in a collection
for elem in collection:
    ...
```

# Patterns for working with collections

```
# number of elements in a collection
len(collection)

# loop over the elements in a collection
for elem in collection:
    ...
```

# Patterns for working with collections

```
# number of elements in a collection
len(collection)

# loop over the elements in a collection
for elem in collection:
    ...

# create a new data structure from an iterable
```

# Patterns for working with collections

```
# number of elements in a collection
len(collection)

# loop over the elements in a collection
for elem in collection:
    ...

# create a new data structure from an iterable
list("abcabc") # => ['a', 'b', 'c', 'a', 'b', 'c']
```

# Patterns for working with collections

```
# number of elements in a collection
len(collection)

# loop over the elements in a collection
for elem in collection:
    ...

# create a new data structure from an iterable
list("abcabc") # => ['a', 'b', 'c', 'a', 'b', 'c']
set("abcabc")  # => {'a', 'b', 'c'}
```

# Patterns for working with collections

```
# number of elements in a collection
len(collection)

# loop over the elements in a collection
for elem in collection:
    ...

# create a new data structure from an iterable
list("abcabc") # => ['a', 'b', 'c', 'a', 'b', 'c']
set("abcabc")  # => {'a', 'b', 'c'}
```

# Patterns for working with collections

```
# number of elements in a collection
len(collection)

# loop over the elements in a collection
for elem in collection:
    ...

# create a new data structure from an iterable
list("abcabc") # => ['a', 'b', 'c', 'a', 'b', 'c']
set("abcabc")  # => {'a', 'b', 'c'}

# enumerate a collection
```

# Patterns for working with collections

```
# number of elements in a collection
len(collection)

# loop over the elements in a collection
for elem in collection:
    ...

# create a new data structure from an iterable
list("abcabc") # => ['a', 'b', 'c', 'a', 'b', 'c']
set("abcabc")  # => {'a', 'b', 'c'}

# enumerate a collection
enumerate(['a', 'b', 41]) # => <(0, 'a'), (1, 'b'), (2, 41)>
```

# Patterns for working with collections

```
# number of elements in a collection
len(collection)

# loop over the elements in a collection
for elem in collection:
    ...

# create a new data structure from an iterable
list("abcabc") # => ['a', 'b', 'c', 'a', 'b', 'c']
set("abcabc")  # => {'a', 'b', 'c'}

# enumerate a collection
enumerate(['a', 'b', 41]) # => <(0, 'a'), (1, 'b'), (2, 41)>
for i, elem in enumerate(['a', 'b', 41]):
```

# Patterns for working with collections

```
# number of elements in a collection
len(collection)

# loop over the elements in a collection
for elem in collection:
    ...

# create a new data structure from an iterable
list("abcabc") # => ['a', 'b', 'c', 'a', 'b', 'c']
set("abcabc")  # => {'a', 'b', 'c'}

# enumerate a collection
enumerate(['a', 'b', 41]) # => <(0, 'a'), (1, 'b'), (2, 41)>
for i, elem in enumerate(['a', 'b', 41]):
    ...
```

# Patterns for working with collections

# Patterns for working with collections

```
# sort a collection
```

# Patterns for working with collections

```
# sort a collection
sorted("cbda")                    # => ['a', 'b', 'c', 'd']
```

# Patterns for working with collections

```
# sort a collection
sorted("cbda")                # => ['a', 'b', 'c', 'd']
sorted("cbda", reverse=True) # => ['d', 'c', 'b', 'a']
```

# Patterns for working with collections

```
# sort a collection
sorted("cbda")                # => ['a', 'b', 'c', 'd']
sorted("cbda", reverse=True) # => ['d', 'c', 'b', 'a']
```

# Patterns for working with collections

```
# sort a collection
sorted("cbda")                # => ['a', 'b', 'c', 'd']
sorted("cbda", reverse=True) # => ['d', 'c', 'b', 'a']

# pairwise combinations
```

# Patterns for working with collections

```
# sort a collection
sorted("cbda")                # => ['a', 'b', 'c', 'd']
sorted("cbda", reverse=True) # => ['d', 'c', 'b', 'a']

# pairwise combinations
zip(
```

# Patterns for working with collections

```
# sort a collection
sorted("cbda")                # => ['a', 'b', 'c', 'd']
sorted("cbda", reverse=True) # => ['d', 'c', 'b', 'a']

# pairwise combinations
zip(
    ['arpit', 'chase', 'will'],
```

# Patterns for working with collections

```
# sort a collection
sorted("cbda")                # => ['a', 'b', 'c', 'd']
sorted("cbda", reverse=True) # => ['d', 'c', 'b', 'a']

# pairwise combinations
zip(
    ['arpit', 'chase', 'will'],
    ['B+', 'A', 'A-']
```

# Patterns for working with collections

```
# sort a collection
sorted("cbda")                # => ['a', 'b', 'c', 'd']
sorted("cbda", reverse=True) # => ['d', 'c', 'b', 'a']

# pairwise combinations
zip(
    ['arpit', 'chase', 'will'],
    ['B+', 'A', 'A-']
) # => <('arpit', 'B+'), ('chase', 'A'), <('will', 'A-')>
```

# Patterns for working with collections

```
# sort a collection
sorted("cbda")                # => ['a', 'b', 'c', 'd']
sorted("cbda", reverse=True) # => ['d', 'c', 'b', 'a']

# pairwise combinations
zip(
    ['arpit', 'chase', 'will'],
    ['B+', 'A', 'A-']
) # => <('arpit', 'B+'), ('chase', 'A'), <('will', 'A-')>
for a, b in zip(collection1, collection2):
```

# Patterns for working with collections

```
# sort a collection
sorted("cbda")                # => ['a', 'b', 'c', 'd']
sorted("cbda", reverse=True) # => ['d', 'c', 'b', 'a']

# pairwise combinations
zip(
    ['arpit', 'chase', 'will'],
    ['B+', 'A', 'A-']
) # => <('arpit', 'B+'), ('chase', 'A'), <('will', 'A-')>
for a, b in zip(collection1, collection2):
    ...
```

# Patterns for working with collections

```
# sort a collection
sorted("cbda")                # => ['a', 'b', 'c', 'd']
sorted("cbda", reverse=True) # => ['d', 'c', 'b', 'a']

# pairwise combinations
zip(
    ['arpit', 'chase', 'will'],
    ['B+', 'A', 'A-']
) # => <('arpit', 'B+'), ('chase', 'A'), <('will', 'A-')>
for a, b in zip(collection1, collection2):
    ...
```

# Patterns for working with collections

```
# sort a collection
sorted("cbda")                # => ['a', 'b', 'c', 'd']
sorted("cbda", reverse=True) # => ['d', 'c', 'b', 'a']

# pairwise combinations
zip(
    ['arpit', 'chase', 'will'],
    ['B+', 'A', 'A-']
) # => <('arpit', 'B+'), ('chase', 'A'), <('will', 'A-')>
for a, b in zip(collection1, collection2):
    ...

# range
```

# Patterns for working with collections

```
# sort a collection
sorted("cbda")                # => ['a', 'b', 'c', 'd']
sorted("cbda", reverse=True) # => ['d', 'c', 'b', 'a']

# pairwise combinations
zip(
    ['arpit', 'chase', 'will'],
    ['B+', 'A', 'A-']
) # => <('arpit', 'B+'), ('chase', 'A'), <('will', 'A-')>
for a, b in zip(collection1, collection2):
    ...

# range
range(a, b, c) # => ints from a (inclusive) to b (exclusive)
```

# Patterns for working with collections

```
# sort a collection
sorted("cbda")                # => ['a', 'b', 'c', 'd']
sorted("cbda", reverse=True) # => ['d', 'c', 'b', 'a']

# pairwise combinations
zip(
    ['arpit', 'chase', 'will'],
    ['B+', 'A', 'A-']
) # => <('arpit', 'B+'), ('chase', 'A'), <('will', 'A-')>
for a, b in zip(collection1, collection2):
    ...

# range
range(a, b, c) # => ints from a (inclusive) to b (exclusive)
               #    with a step size of c
```

# Patterns for working with collections

```
# sort a collection
sorted("cbda")                # => ['a', 'b', 'c', 'd']
sorted("cbda", reverse=True) # => ['d', 'c', 'b', 'a']

# pairwise combinations
zip(
    ['arpit', 'chase', 'will'],
    ['B+', 'A', 'A-']
) # => <('arpit', 'B+'), ('chase', 'A'), <('will', 'A-')>
for a, b in zip(collection1, collection2):
    ...

# range
range(a, b, c) # => ints from a (inclusive) to b (exclusive)
               #    with a step size of c
range(3, 10, 2) # => <3, 5, 7, 9>
```

# Comprehensions

Write a function that returns a list of all odd square numbers below 100

`odd_squares.py`

# Comprehensions

Write a function that returns a list of all odd square numbers below 100

odd_squares.py

```
for i in range(loop_max):
    if (i ** 2) % 2 != 0:
        output.append(i ** 2)
```

# Comprehensions

Write a function that returns a list of all odd square numbers below 100

odd_squares.py

```
for i in range(loop_max):
    if (i ** 2) % 2 != 0:
        output.append(i ** 2)
```

Go through a collection...

...check some condition...

...apply some operation to the element.

# Comprehensions

Write a function that returns a list of all odd square numbers below 100

odd_squares.py

```
for i in range(loop_max):
    if (i ** 2) % 2 != 0:
        output.append(i ** 2)
```

Go through a collection...

...check some condition...

...apply some operation to the element.

# Comprehensions

Write a function that returns a list of all odd square numbers below 100

odd_squares.py

```
return [
    i ** 2
    for i in range(int(num ** (1/2)))
    if (i ** 2) % 2 != 0
]
```

...apply some operation to the element.

Go through a collection...

...check some condition...

# Comprehensions

```
[fn(x) for x in iterable]
```

# Comprehensions

```
[fn(x) for x in iterable if cond(x)]
```

# Comprehensions

Square brackets define a list.

```
[fn(x) for x in iterable if cond(x)]
```

# Comprehensions

Square brackets define a list.

```
[fn(x) for x in iterable if cond(x)]
```

Apply this function...

# Comprehensions

Square brackets define a list.

```
[fn(x) for x in iterable if cond(x)]
```

Apply this function...

...to each element of this iterable...

# Comprehensions

Square brackets define a list.

```
[fn(x) for x in iterable if cond(x)]
```

Apply this function...

...to each element of this iterable...

...when this condition holds.

# Comprehensions

```
{f(k):g(v) for k, v in iterable if cond(k, v)}
```

# Comprehensions

Curly brackets, colon denote a dictionary!

```
{f(k):g(v) for k, v in iterable if cond(k, v)}
```

# Comprehensions

Curly brackets, colon denote a dictionary!

```
{f(k):g(v) for k, v in iterable if cond(k, v)}
```