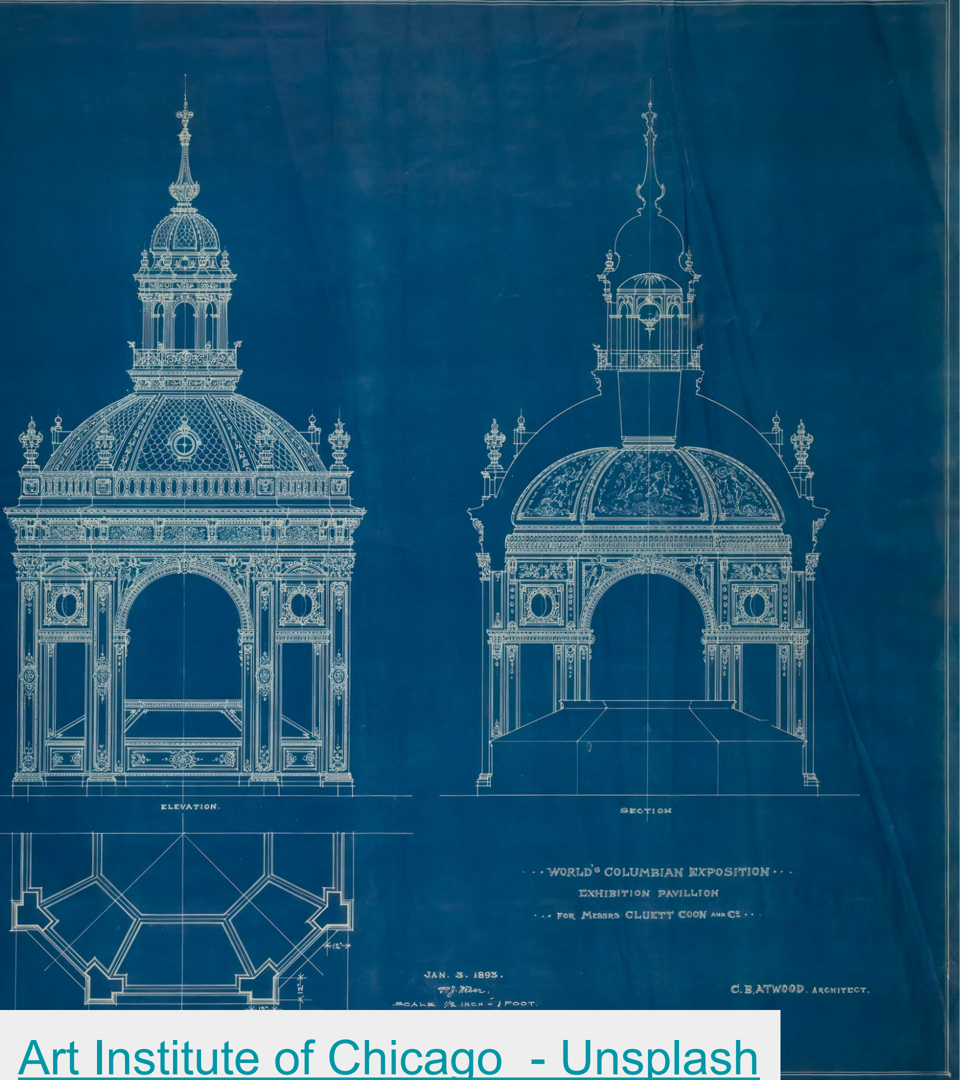


Art Institute of Chicago - Unsplash

Classes

Objects

Python is an *object-oriented* language

Objects

Python is an *object-oriented* language

We typically think about the objects (things) that we are representing in our code

Types of data

What data types have we talked about in this class?

Types of data

What data types have we talked about in this class?

Basic data types:

- int
- float
- str
- bool

Types of data

What data types have we talked about in this class?

Basic data types:

- int
- float
- str
- bool

Container data types:

- list
- tuple
- dict

Storing complex objects

How can we store something complex, like a car, a game character, or an artist on Spotify?

Storing complex objects

How can we store something complex, like a car, a game character, or an artist on Spotify?

We *can* collect multiple pieces of data in a container (list / dictionary)

Storing complex objects

How can we store something complex, like a car, a game character, or an artist on Spotify?

We *can* collect multiple pieces of data in a container (list / dictionary)

What are the down sides?

Storing complex objects

How can we store something complex, like a car, a game character, or an artist on Spotify?

We *can* collect multiple pieces of data in a container (list / dictionary)

What are the down sides?

- How do we manipulate these complex structures? It's a pain!

Storing complex objects

How can we store something complex, like a car, a game character, or an artist on Spotify?

We *can* collect multiple pieces of data in a container (list / dictionary)

What are the down sides?

- How do we manipulate these complex structures? It's a pain!
- It's difficult to separate the pieces!

Breaking down the pieces

Imagine we want to store an artist's information on a platform like Spotify

Breaking down the pieces

Imagine we want to store an artist's information on a platform like Spotify

What data do you need to store?

How do you store it?

Breaking down the pieces

Imagine we want to store an artist's information on a platform like Spotify

An example:

Breaking down the pieces

Imagine we want to store an artist's information on a platform like Spotify

An example:

Each artist is represented by a dictionary.

Breaking down the pieces

Imagine we want to store an artist's information on a platform like Spotify

An example:

Each artist is represented by a dictionary.

Songs are tracked in a list, with each song being a dictionary of name, length, and the actual audio data.

Breaking down the pieces

Imagine we want to store an artist's information on a platform like Spotify

An example:

Each artist is represented by a dictionary.

Songs are tracked in a list, with each song being a dictionary of name, length, and the actual audio data.

Albums are stored in a list, with each album being a dictionary that holds the name, song list, album art, etc.

Breaking down the pieces

Imagine we want to store an artist's

What does it look like to add a new song?

potify

An example:

Each artist is represented by a dictionary.

Songs are tracked in a list, with each song being a dictionary of name, length, and the actual audio data.

Albums are stored in a list, with each album being a dictionary that holds the name, song list, album art, etc.

One step further

We want to break out objects down into smaller pieces

One step further

We want to break out objects down into smaller pieces

But we also want these pieces to be easily manageable…

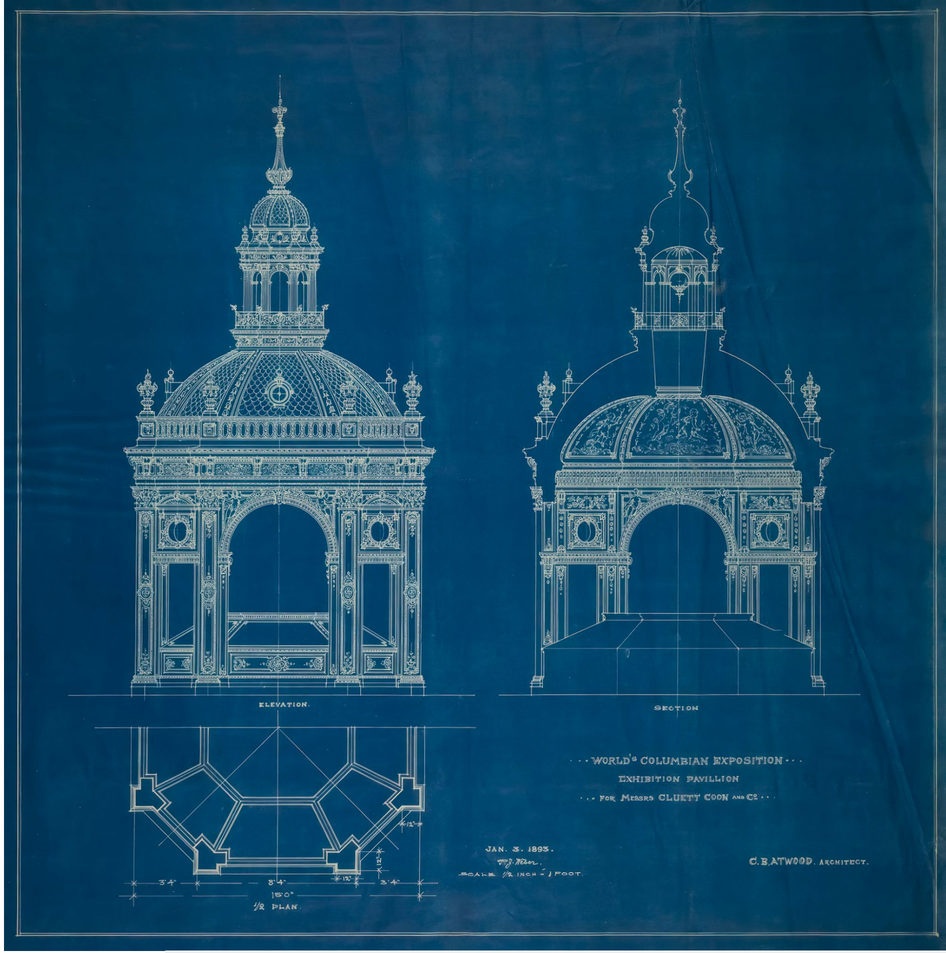
One step further

We want to break out objects down into smaller pieces

But we also want these pieces to be easily manageable…

Classes allow us to store data AND have functions to manage that data!

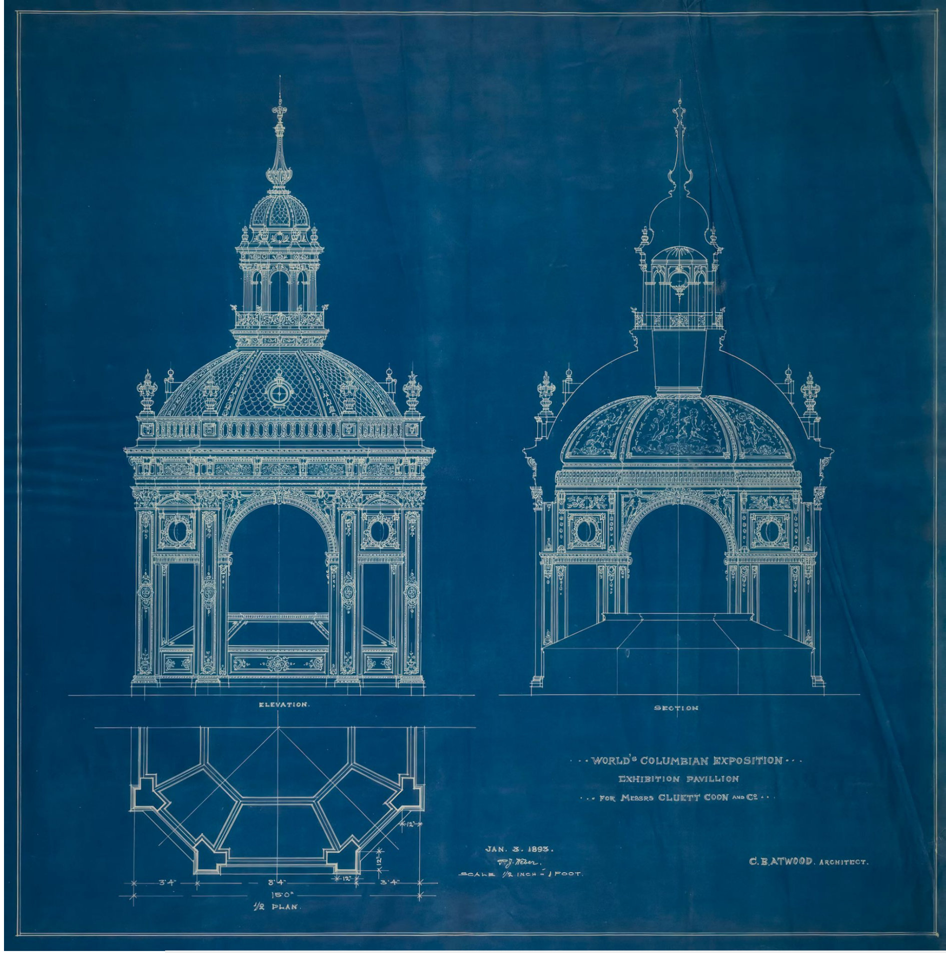
Terminology



[Art Institute of Chicago - Unsplash]

Terminology

Think of a <u>class</u> as a blueprint

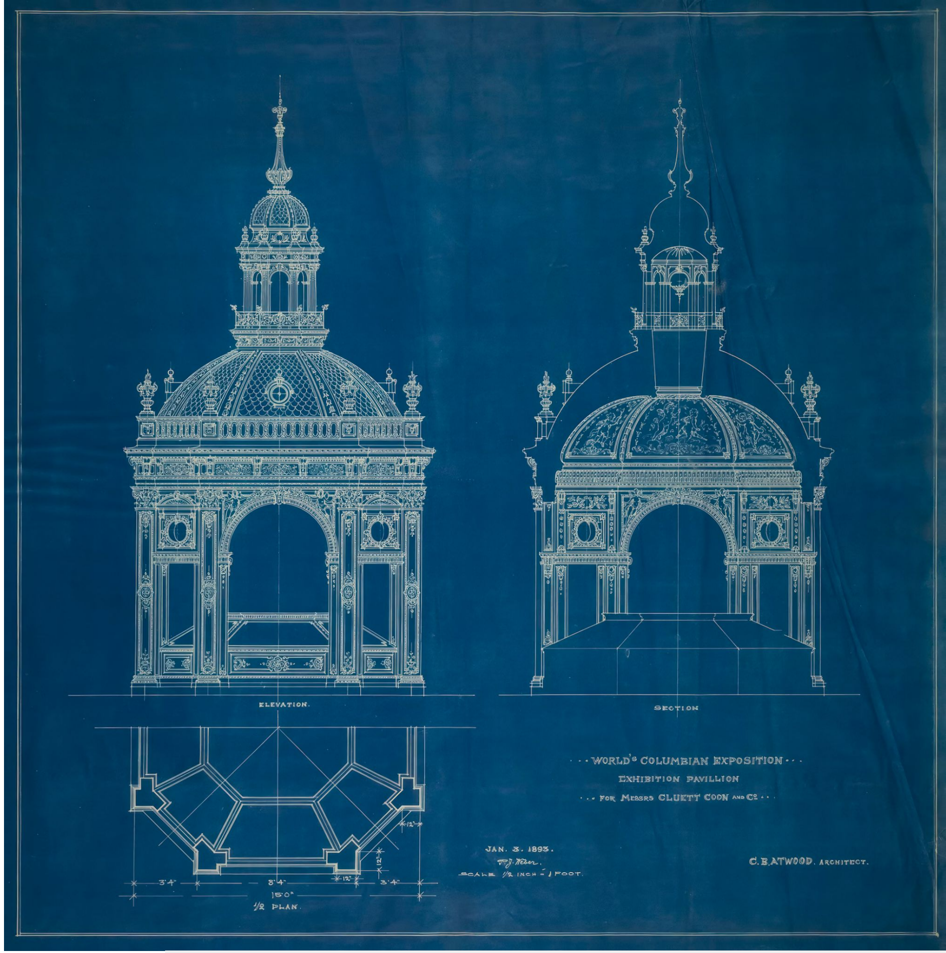


[Art Institute of Chicago - Unsplash]

Terminology

Think of a class as a blueprint

We can create *instances* of this class



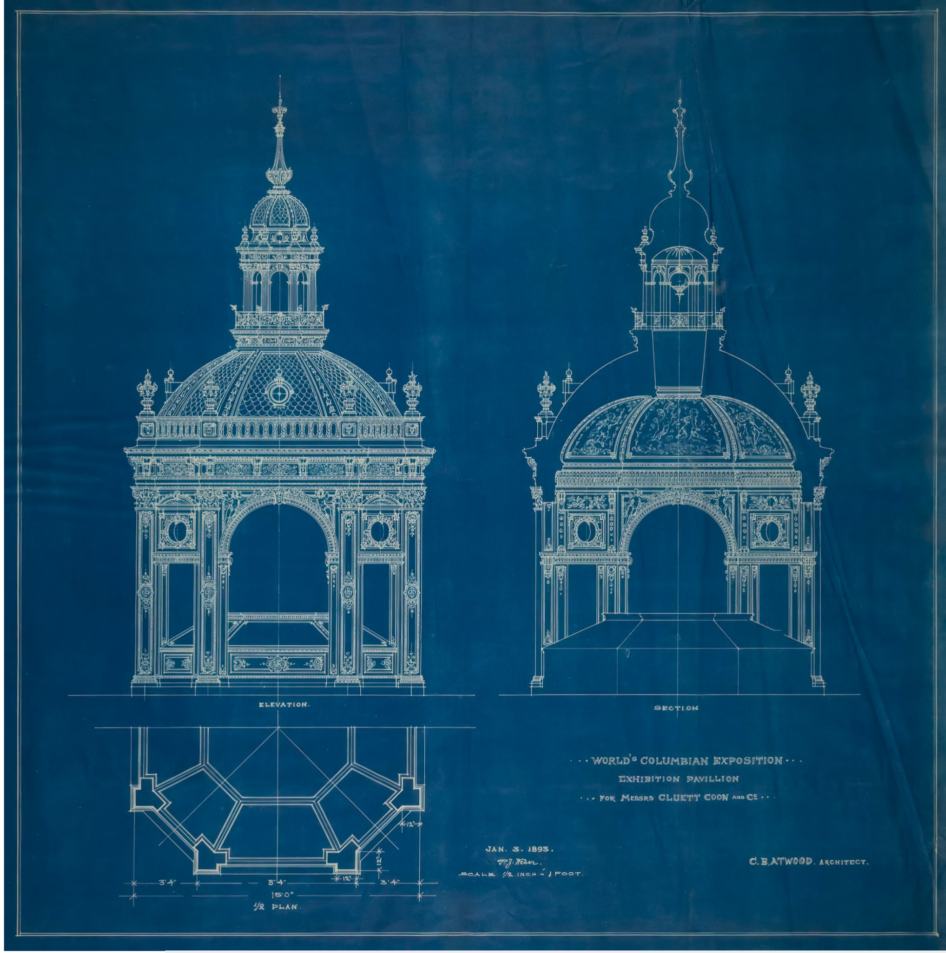
Art Institute of Chicago - Unsplash

Terminology

Think of a class as a blueprint

We can create *instances* of this class

Python makes a new *object*
following the blueprint



[Art Institute of Chicago - Unsplash]

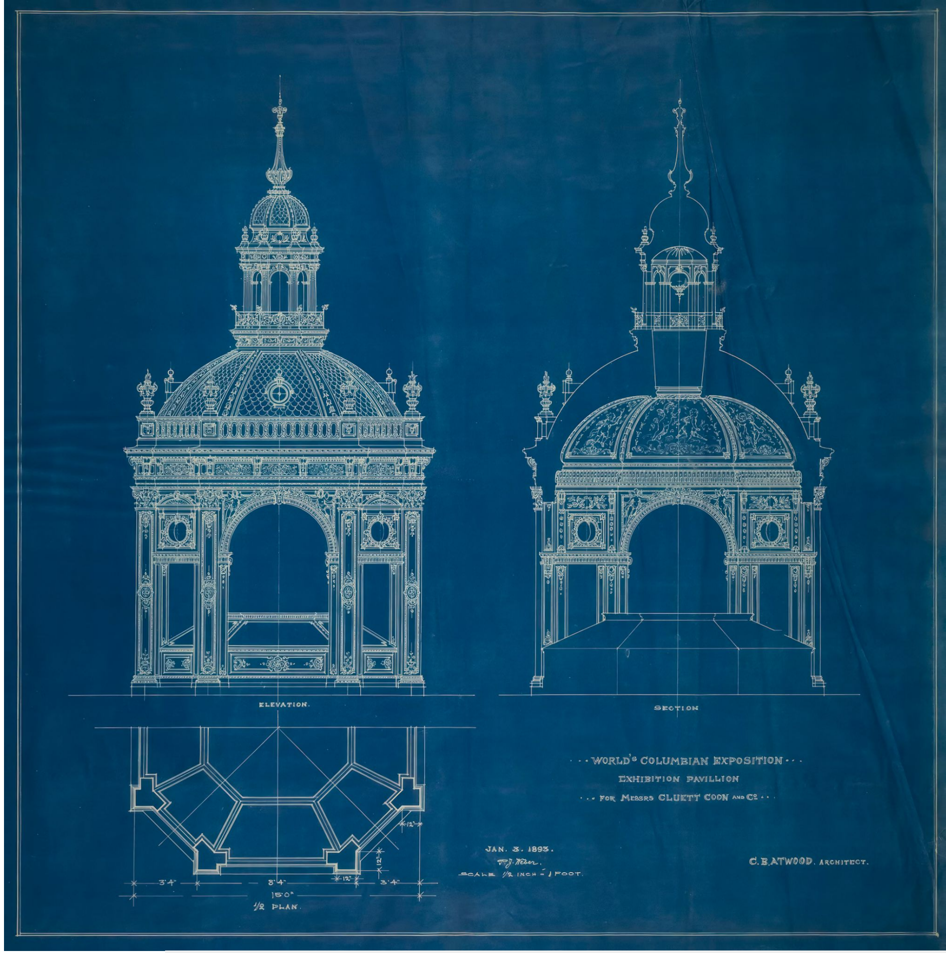
Terminology

Think of a class as a blueprint

We can create *instances* of this class

Python makes a new *object*
following the blueprint

This process is called *instantiation*



Art Institute of Chicago - Unsplash

You've seen this before!

We have already been working with objects that have methods

Reminder: a method is a function associated with a particular object

Strings have methods! `str.lower(), str.count(), str.isdigit()`

Our project 2 is using objects from PIL!

`draw.rectangle()` is a method attached to an ImageDraw.Draw object!

Why methods?

Why methods?

Dillon Kidd - Unsplash

Why methods?

Cole Freeman - Unsplash

Dillon Kidd - Unsplash

Why me

Dillon Kidd - Unsplash

Anatolii Nesterov - Unsplash

Cole Freeman - Unsplash

Interfaces

One of the biggest benefits of classes is to create an *interface*

Interfaces

One of the biggest benefits of classes is to create an *interface*

The simplified way in which *others* interact with your code

Interfaces

One of the biggest benefits of classes is to create an *interface*

The simplified way in which *others* interact with your code

Interface of a car: steering wheel, pedals, shifter

Interfaces

One of the biggest benefits of classes is to create an *interface*

The simplified way in which *others* interact with your code

Interface of a car: steering wheel, pedals, shifter
Interface of the mouse in a maze: rotate_left, rotate_right, move

Defining a class

Much like functions, we can *define* a class

Defining a class

Much like functions, we can *define* a class

This tells Python what the class is and what it does

Defining a class

Much like functions, we can *define* a class

This tells Python what the class is and what it does

```
class Coin:
   # Details of class go in here!
```

Defining a class

Much like functions, we can *define* a class

Defining a class means we are defining a few things:

1. The class's name
2. The *methods* (functions) associated with this class
3. The *attributes* (variables) associated with this class

Defining the methods of a class

Methods are functions, so you
know the general form!

Defining the methods of a class

Methods are functions, so you know the general form!

```
class Dog:
```

Defining the methods of a class

Methods are functions, so you know the general form!

```
class Dog:
   def name(args):
      # function body
```

Defining the methods of a class

```
class Dog:
    def name(args):
        # function body
```

Methods are functions, so you know the general form!

All methods should use `self` as their first argument

Defining the methods of a class

Methods are functions, so you know the general form!

```
class Dog:
   def name(self, args):
      # function body
```

All methods should use `self` as their first argument

What is self?

Imagine we have this class:

What is self?

Imagine we have this class:

```
class NPC:
```

What is self?

Imagine we have this class:

```
class NPC:
    def greet(self):
        print(f"Hello! My name is {self.name}")
```

What is self?

Imagine we have this class:

```
class NPC:
    def greet(self):
        print(f"Hello! My name is {self.name}")
```

And we have two instances of NPC: `alice` and `bob`

What is self?

Imagine we have this class:

```
class NPC:
    def greet(self):
        print(f"Hello! My name is {self.name}")
```

And we have two instances of NPC: `alice` and `bob`

If I run this code, how does python know who is speaking?

What is self?

Imagine we have this class:

```
class NPC:
    def greet(self):
        print(f"Hello! My name is {self.name}")
```

And we have two instances of NPC: `alice` and bob

If I run this code, how does python know who is speaking?

```
alice.greet()
bob.greet()
```

What is self?

Imagine we have this class:

```
class NPC:
   def greet(self):
      print(f"Hello! My name is {self.name}")
```

And we have two instances of NPC: `alice` and `bob`

If I run this code, how does python know who is speaking?

```
alice.greet() # For alice, self is alice
bob.greet()   # For bob, self is bob
```

What is self?

`self` refers to the current instance of our class

Imagine we have this class:

```
class NPC:
    def greet(self):
        print(f"Hello! My name is {self.name}")
```

And we have two instances of NPC: `alice` and `bob`

If I run this code, how does python know who is speaking?

```
alice.greet() # For alice, self is alice
bob.greet()   # For bob, self is bob
```

Back to methods

`self` must be the first argument

```
class Dog:
   def name(self, args):
      # function body
```

Back to methods

`self` must be the first argument

Other than self, methods are just normal functions!

```
class Dog:
   def name(self, args):
      # function body
```

Back to methods

`self` must be the first argument

Other than self, methods are just normal functions!

You can have other arguments after `self`

```
class Dog:
   def name(self, args):
      # function body
```

Back to methods

`self` must be the first argument

Other than self, methods are just normal functions!

You can have other arguments after `self`

```
class Dog:
    def dig(self, depth, rad):
        # function body
```

Back to methods

`self` must be the first argument

Other than self, methods are just normal functions!

You can have other arguments after `self`

Methods can also *return* values

```
class Dog:
    def dig(self, depth, rad):
        # function body
```

Back to methods

`self` must be the first argument

Other than self, methods are just normal functions!

You can have other arguments after `self`

Methods can also *return* values

```
class Dog:
    def dig(self, depth, rad):
        # function body

    def get_human_age(self):
        return self.age * 7
```

Back to methods

`self` must be the first argument

Other than self, methods are just normal functions!

You can have other arguments after `self`

Methods can also *return* values

```
class Dog:
    def dig(self, depth, rad):
        # function body

    def get_human_age(self):
        return self.age * 7
```

```
fido = Dog()
fido_age = fido.get_human_age()
```

The `init` method

Classes in Python have a few special methods

The `init` method

Classes in Python have a few special methods

- These methods begin and end with two underscores (`__`)

The `init` method

Classes in Python have a few special methods

- These methods begin and end with two underscores (`__`)

For now, we'll focus on the `__init__` method

The `init` method

Classes in Python have a few special methods

- These methods begin and end with two underscores (`__`)

For now, we'll focus on the `__init__` method

- `__init__` is called when we create an instance of our class

The `init` method

Classes in Python have a few special methods

These methods begin and end with two underscores (`__`)

For now, we'll focus on the `__init__` method

`__init__` is called when we create an instance of our class

This is where we'll initialize (setup) our instance!

An init example

```
class Plant:
   def __init__(self, species):
      print(f'Species: {species}')
```

An init example

```
class Plant:
   def __init__(self, species):
      print(f'Species: {species}')


plant_1 = Plant("rose")
plant_2 = Plant("bamboo")
```

An init example

```
class Plant:
   def __init__(self, species):
      print(f'Species: {species}')


plant_1 = Plant("rose")
plant_2 = Plant("bamboo")
```

What is the output?

An init example

```
class Plant:
   def __init__(self, species):
      print(f'Species: {species}')
```

```
plant_1 = Plant("rose")
plant_2 = Plant("bamboo")
```

What is the output?

```
rose
bamboo
```

Attributes

Attributes are the data that we store in each instance of a class

Attributes

Attributes are the data that we store in each instance of a class

These are usually set in the init method (but can be done elsewhere)

```
class Plant:
   def __init__(self, species_name):
      self.species = species_name
```

Attributes

Attributes are the data that we store in each instance of a class

These are usually set in the init method (but can be done elsewhere)

```
class Plant:
    def __init__(self, species_name):
        self.species = species_name
```

This is setting the species attribute of our Plant instance!

Attributes

Attributes are the data that we store in each instance of a class

These are usually set in the init method (but can be done elsewhere)

```
class Plant:
   def __init__(self, species):
       self.species = species
```

This is setting the species attribute of our Plant instance!

Attributes

Attributes are the data that we store in each instance of a class

- These are usually set in the init method (but can be done elsewhere)

```
class Plant:
    def __init__(self, species):
        self.species = species
```

This is setting the species attribute of our Plant instance!

Attributes

Attributes are the data that we store in each instance of a class

These are usually set in the init method (but can be done elsewhere)

```
class Plant:
    def __init__(self, species):
        self.species = species
```

This is setting the species attribute of our Plant instance!

We can access attributes in other methods!

Attributes example

```
class Plant:
    def __init__(self, species):
        self.species = species
        self.height = 0
```

Attributes example

```
class Plant:
   def __init__(self, species):
      self.species = species
      self.height = 0

   def grow(self, amount):
      self.height += amount
```

Attributes example

```
class Plant:
   def __init__(self, species):
      self.species = species
      self.height = 0

   def grow(self, amount):
      self.height += amount

   def print_info(self):
      print(f'Species: {self.species}')
      print(f'Height: {self.height} inches')
```

More special methods

We've talked about `__init__`, but there are many!

More special methods

We've talked about `__init__`, but there are many!

`__lt__`, `__gt__`, `__leq__`, `__eq__`, etc for comparison

More special methods

We've talked about `__init__`, but there are many!

`__lt__, __gt__, __leq__, __eq__,` etc for comparison

`__add__, __sub__, __mul__,` etc for math

More special methods

We've talked about `__init__`, but there are many!

`__lt__`, `__gt__`, `__leq__`, `__eq__`, etc for comparison

`__add__`, `__sub__`, `__mul__`, etc for math

`__int__`, `__str__`, `__float__` for type conversions

Example: str and add

Imagine I have the following class:

```
class Vector2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Example: str and add

Imagine I have the following class:

```
class Vector2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

How do I create an instance of our class?

Example: str and add

Imagine I have the following class:

```
class Vector2D:
   def __init__(self, x, y):
      self.x = x
      self.y = y

v = Vector2D(1, 2)
w = Vector2D(3, -1)
```

How do I create an instance of our class?

Example: str and add

Imagine I have the following class:

```
class Vector2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y

v = Vector2D(1, 2)
w = Vector2D(3, -1)
```

How do I create an instance of our class?

What should it look like when I print a vector?

Example: str and add

Imagine I have the following class:

```
class Vector2D:
   def __init__(self, x, y):
      self.x = x
      self.y = y

v = Vector2D(1, 2)
w = Vector2D(3, -1)
```

How do I create an instance of our class?

What should it look like when I print a vector?

What would I add to a vector?
What would that look like?

Instance vs class scope

We've talked about *instance* variables (attributes)

Instance vs class scope

We've talked about *instance* variables (attributes)

`self.x` is an attribute of our instance

Instance vs class scope

We've talked about *instance* variables (attributes)

`self.x` is an attribute of our instance

We can also store information in the class that *all* instances share

Instance vs class scope

We've talked about *instance* variables (attributes)

`self.x` is an attribute of our instance

We can also store information in the class that *all* instances share

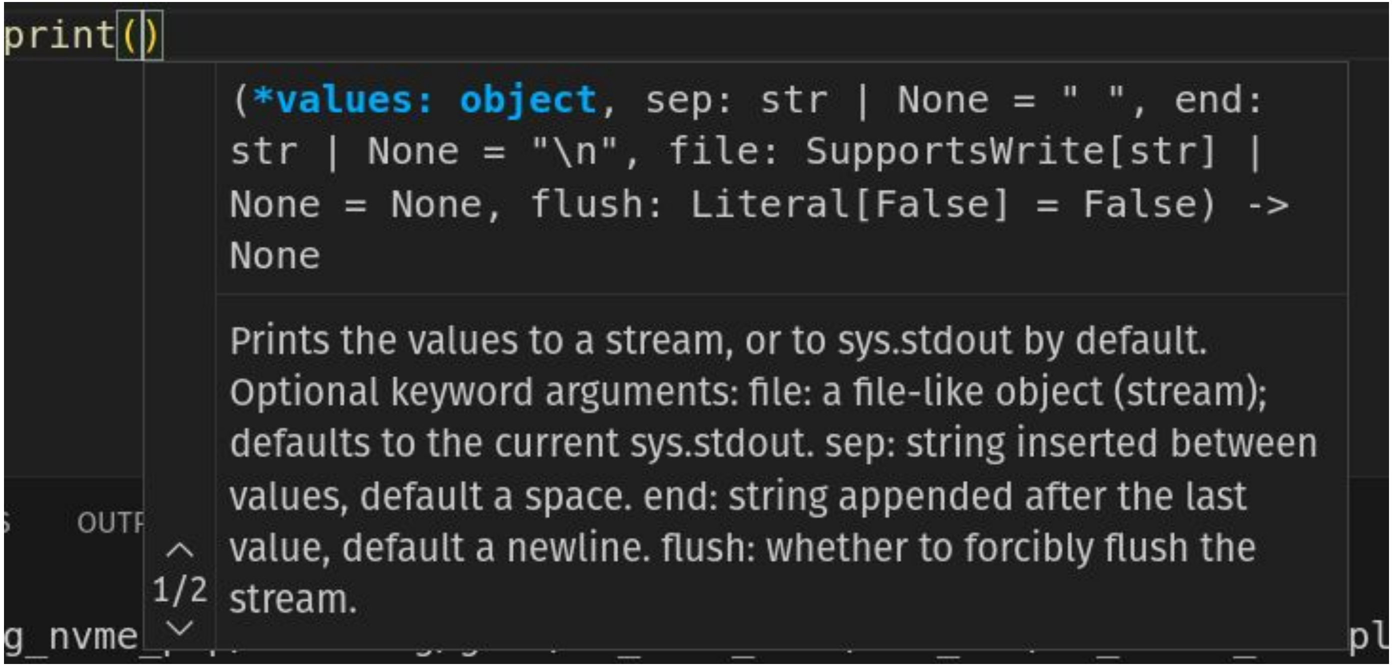
```
class Part:
    next_id  = 1

    def __init__(self, name):
        self.name = name
        self.id = Part.next_id
        Part.next_id += 1
```

docstrings

Have you noticed that VSCode will give you information about the functions you are calling?

docstrings

Have you noticed that VSCode will give you information about the functions you are calling?



```
print()
```

```
(*values: object, sep: str | None = " ", end: str | None = "\n", file: SupportsWrite[str] | None = None, flush: Literal[False] = False) -> None
```

Prints the values to a stream, or to sys.stdout by default. Optional keyword arguments: file: a file-like object (stream); defaults to the current sys.stdout. sep: string inserted between values, default a space. end: string appended after the last value, default a newline. flush: whether to forcibly flush the stream.

docstrings

We want to write that documentation in a way that VSCode can use

docstrings

We want to write that documentation in a way that VSCode can use

Quick aside: How do strings handle newlines?

docstrings

We want to write that documentation in a way that VSCode can use

Quick aside: How do strings handle newlines?

```
str_1 = "This is\nvalid"
```

docstrings

We want to write that documentation in a way that VSCode can use

Quick aside: How do strings handle newlines?

```
str_1 = "This is\nvalid"
str_2 = "This is
not"
```

docstrings

We want to write that documentation in a way that VSCode can use

Quick aside: How do strings handle newlines?

```
str_1 = "This is\nvalid"
str_2 = "This is
not"
```

What if we want a string to have a newline AND show it in our code?

docstrings

We want to write that documentation in a way that VSCode can use

Quick aside: How do strings handle newlines?

```
str_1 = "This is\nvalid"
str_2 = "This is
not"
str_3 = """This is all
one valid string.
Yay!"""
```

What if we want a string to have a newline AND show it in our code?

docstrings

We can use these multiline strings to add that documentation!

docstrings

We can use these multiline strings to add that documentation!

To create a docstring, add a string as the first line in a function/class!

docstrings

We can use these multiline strings to add that documentation!

To create a docstring, add a string as the first line in a function/class!

```
def func():
    """This is a docstring!"""
    pass
```

docstrings

More info on proper docstrings here: https://peps.python.org/pep-0257/

Type hinting

What's the type of x?

```
x = 7.2
```

Type hinting

What's the type of x? A float!

```
x = 7.2
```

Type hinting

What's the type of x? A float!

```
x = 7.2
```

What if our code told us that directly?

Type hinting

What's the type of x? A float!

```
x = 7.2
```

What if our code told us that directly?

That's what type hinting does!

Type hinting

What's the type of x? A float!

```
x = 7.2
```

What if our code told us that directly?

That's what type hinting does!

```
x : float = 7.2
```

Type hinting

What's the type of x? A float!

```
x = 7.2
```

What if our code told us that directly?

That's what type hinting does!

```
x : float = 7.2
```

Type hinting cheat sheet:

https://mypy.readthedocs.io/en/stable/cheat_sheet_py3.html

Type hinting

What happens when we run this code?

```
x : int = 5
x = 4.5
print(x)
```

Type hinting

What happens when we run this code?

```
x : int = 5
x = 4.5
print(x)
```

4.5 is printed, no errors!

Type hinting

What happens when we run this code?

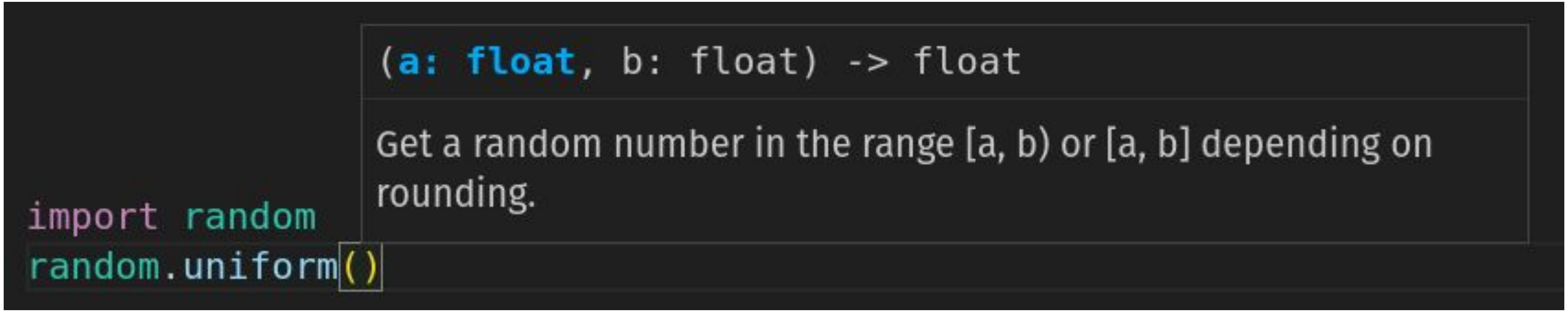
```
x : int = 5
x = 4.5
print(x)
```

4.5 is printed, no errors!

Type hints are NOT rules! They are not enforced

Type hinting - Why?

Type hinting - Why?

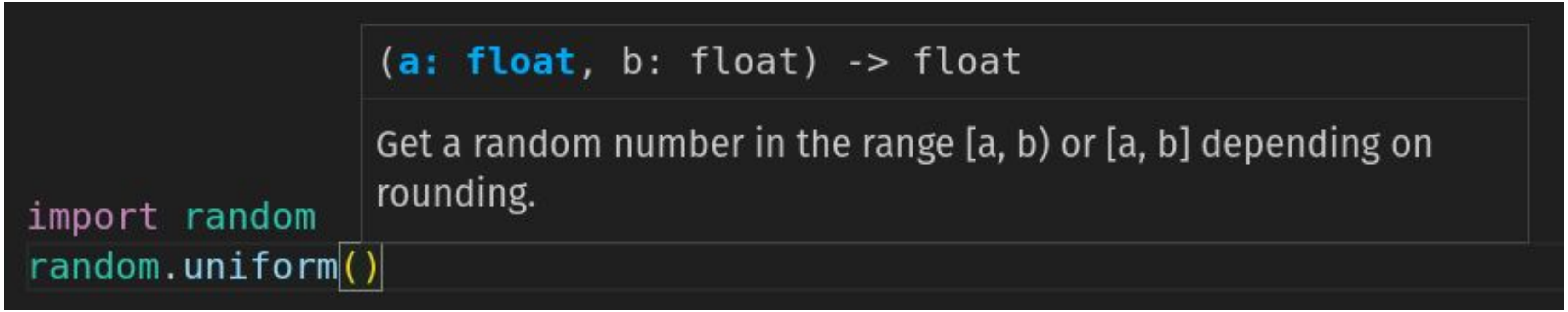


```
(a: float, b: float) -> float
```

Get a random number in the range [a, b) or [a, b] depending on rounding.

```
import random
random.uniform()
```

Type hinting - Why?



```
(a: float, b: float) -> float
```

Get a random number in the range [a, b) or [a, b] depending on rounding.

```python
import random
random.uniform()
```

Type hints are used like docstrings to tell you what a function takes and returns!

Type hinting in functions

Type hinting in functions

Arguments: use the same syntax as before

Type hinting in functions

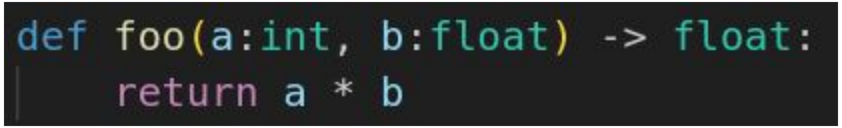
Arguments: use the same syntax as before

Return type: an arrow (->) pointing to the return type just before the colon

Type hinting in functions

Arguments: use the same syntax as before

Return type: an arrow (->) pointing to the return type just before the colon

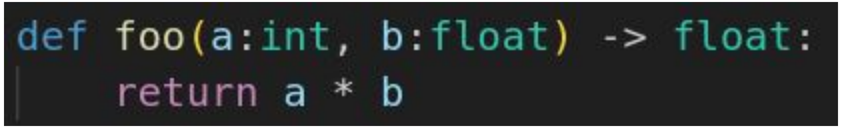


```python
def foo(a:int, b:float) -> float:
    return a * b
```

Type hinting: complex types

Arguments: use the same syntax as before

Return type: an arrow (->) pointing to the return type just before the colon



```
def foo(a:int, b:float) -> float:
    return a * b
```

Type hinting: complex types

A list can be:

```
my_list: list
my_list: list[int]
```

Type hinting: complex types

A list can be:

```
my_list: list
my_list: list[int]
```

Dictionaries:

```
D: dict
D: dict[str, int]
```

Type hinting: complex types

A list can be:

```
my_list: list
my_list: list[int]
```

Dictionaries:

```
D: dict
D: dict[str, int]
```

If using a custom class, just use the class name!