Workshop: How to Java in Python

Erwin de Wolff

Radboud University

Overview

- Why use Python?
- Why use OO?
- How to: X
- Questions & Practical

General Comments

- I do not dislike Java
- I’m **not** a professional Python user
- Demonstrations use python 3, and pygame in the demo
- This workshop assumes some understanding of Object Oriented Programming
  - Let me know if this knowledge is ‘in development’!

Why use Python?

- No explicit typing allows for quick scripts
  - But can also cause confusion...

- Python supports many programming paradigms (iterative, OO, functional)
  - List comprehensions are good!

- Code reads almost like pseudocode
  - ‘a != b’ becomes ‘not a is b’
  - ‘list.contains(x)’ becomes ‘x in list’

- Great deal of libraries/packages for AI
  - Numpy, Scipy, Pytorch, Tensorflow and many more!

- It’s free!

Why use OO?

- Classes assemble information and functions into relevant structures
- This helps implement in a top-down manner
- Many ‘things’ have shared information, this can be exploited for efficiency

So why no Java?

- Java takes a lot longer to code
  - Quick scripts are more suited to python's free style
- Some research packages only exist for python, and not java
- Java does not allow for some of python's features like functional programming

So why no Java?

- Java takes a lot longer to code
  - Quick scripts are more suited to python’s free style

- Some research packages only exist for python, and not java

- Java does not allow for some of python’s features like functional programming
  - Except that it does… a bit
  - Java has lambda expressions and map functions

How to: Objects

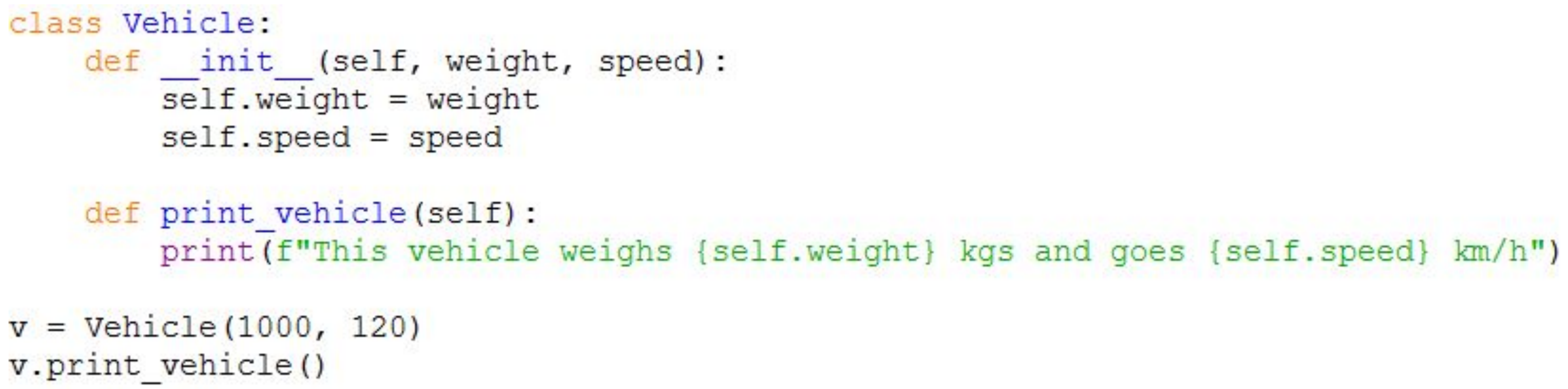
- The goal is to allow us to make objects and call functions on them

- Python already does this!
  - ‘“A string”.split(“ “)’ is an example of calling a function on a string.
  - In that sense, a string is an object of text (albeit an unusual one).

- Generally objects require a class to be defined

How to: Classes

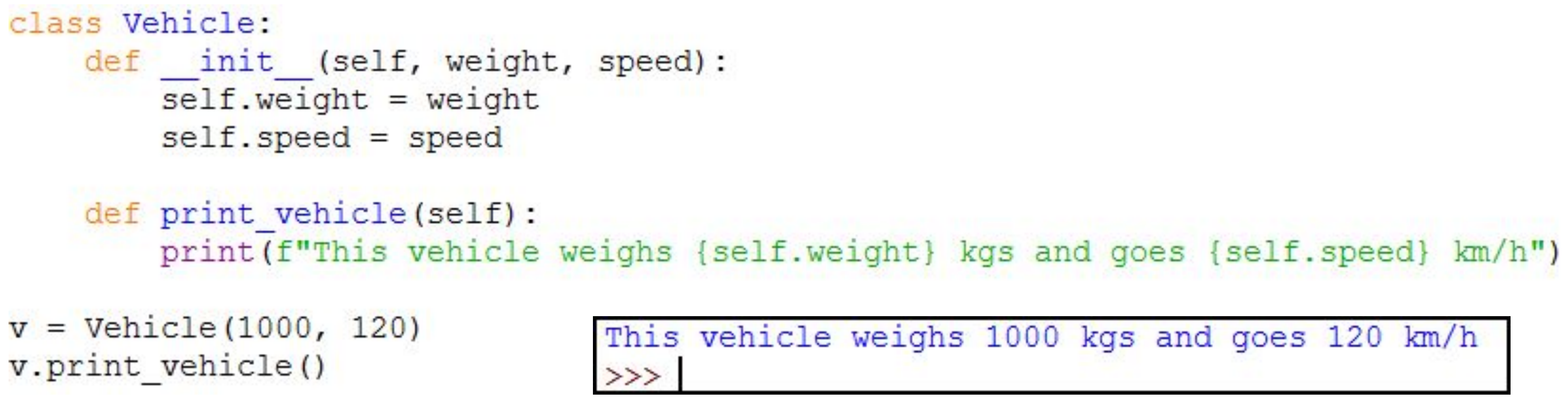
- A class in Python uses a systematic set-up:
- Keywords: *class*, ***self***, *def __init__*



```
class Vehicle:
    def __init__(self, weight, speed):
        self.weight = weight
        self.speed = speed

    def print_vehicle(self):
        print(f"This vehicle weighs {self.weight} kgs and goes {self.speed} km/h")

v = Vehicle(1000, 120)
v.print_vehicle()
```

How to: Classes

- A class in Python uses a systematic set-up:
- Keywords: *class*, ***self***, *def __init__*



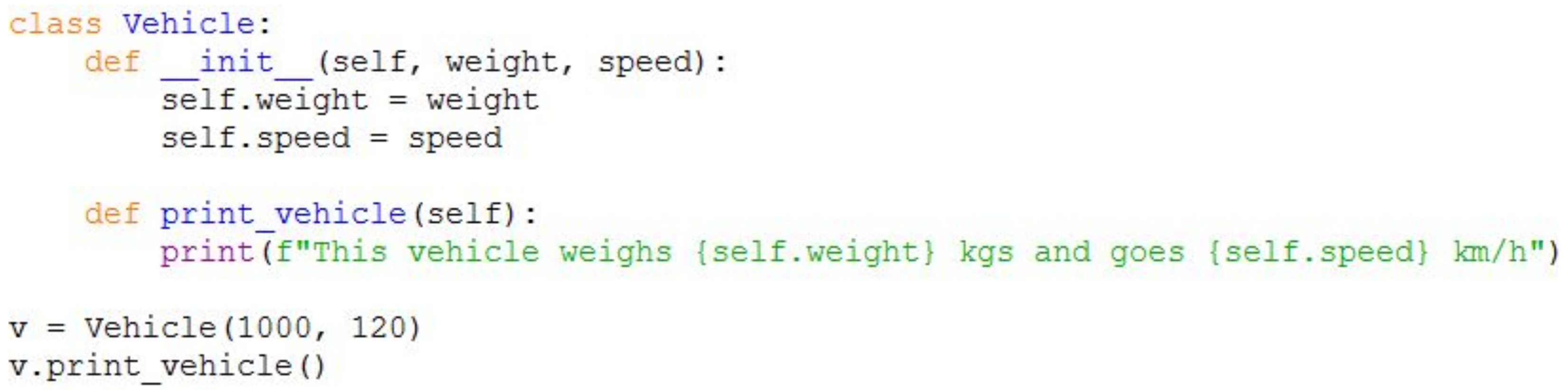
```
class Vehicle:
    def __init__(self, weight, speed):
        self.weight = weight
        self.speed = speed

    def print_vehicle(self):
        print(f"This vehicle weighs {self.weight} kgs and goes {self.speed} km/h")

v = Vehicle(1000, 120)
v.print_vehicle()
```

```
This vehicle weighs 1000 kgs and goes 120 km/h
>>> 
```

How to: Classes

- You do not need a constructor
  - Classes can be a like shed with tools (methods)

- Method attributes are accessed through the ‘self.’ prefix
  - Much like java’s ‘this.’, but always required



```python
class Vehicle:
    def __init__(self, weight, speed):
        self.weight = weight
        self.speed = speed

    def print_vehicle(self):
        print(f"This vehicle weighs {self.weight} kgs and goes {self.speed} km/h")

v = Vehicle(1000, 120)
v.print_vehicle()
```

How to: Inheritance

- Classes allow for inheritance:
  - If an object X is exactly like Y, but different in a few ways.
- Two options for constructor:
  - ‘super’ as shown here
  - ‘Bird.__init__(self, “Ostrich”)’
- **Q:** What happens when we ask:

```python
class Bird:
    def __init__(self, name):
        self.name = name

    def has_feathers(self):
        print(f"{self.name} has feathers")

    def can_fly(self):
        print(f"{self.name} can fly")

    def lays_eggs(self):
        print(f"{self.name} lays eggs")


class Ostrich(Bird):
    def __init__(self):
        super(Ostrich, self).__init__("Ostrich")

    def can_fly(self):
        print(f"Ostrich can NOT fly")
```

How to: Inheritance

- Classes allow for inheritance:
  - If an object X is exactly like Y, but different in a few ways.
- Two options for constructor:
  - ‘super’ as shown here
  - ‘Bird.__init__(self, “Ostrich”)’
- **Q:** What happens when we ask:
  - ‘Ostrich().can_fly()’?

```python
class Bird:
    def __init__(self, name):
        self.name = name

    def has_feathers(self):
        print(f"{self.name} has feathers")

    def can_fly(self):
        print(f"{self.name} can fly")

    def lays_eggs(self):
        print(f"{self.name} lays eggs")


class Ostrich(Bird):
    def __init__(self):
        super(Ostrich, self).__init__("Ostrich")

    def can_fly(self):
        print(f"Ostrich can NOT fly")
```

How to: Inheritance

- Classes allow for inheritance:
  - If an object X is exactly like Y, but different in a few ways.
- Two options for constructor:
  - ‘super’ as shown here
  - ‘Bird.__init__(self, “Ostrich”)’
- **Q:** What happens when we ask:
  - ‘Ostrich().can_fly()’?
  - ‘Ostrich().lays_eggs()’?

```python
class Bird:
    def __init__(self, name):
        self.name = name

    def has_feathers(self):
        print(f"{self.name} has feathers")

    def can_fly(self):
        print(f"{self.name} can fly")

    def lays_eggs(self):
        print(f"{self.name} lays eggs")


class Ostrich(Bird):
    def __init__(self):
        super(Ostrich, self).__init__("Ostrich")

    def can_fly(self):
        print(f"Ostrich can NOT fly")
```

How to: Attribute Protection

- Python attributes are public by default

- To make an attribute *protected*, add a prefix _ to it
  - This makes the attribute accessible only to itself and sub-classes

- To make an attribute *private*, add a prefix __ to it
  - This makes the attribute accessible only to itself

How to: Attribute Protection

- Python attributes are public by default

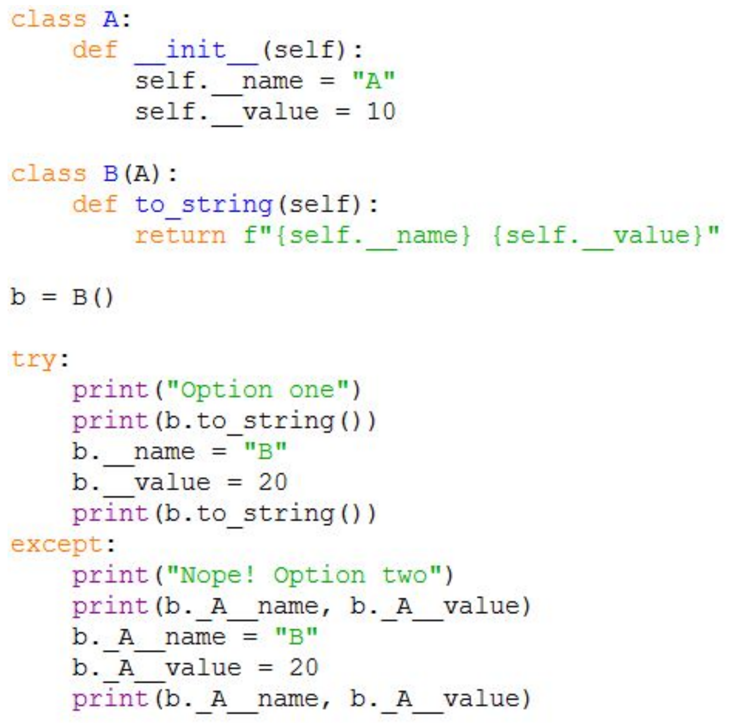
- To make an attribute *protected*, add a prefix _ to it
  - This makes the attribute accessible only to itself and sub-classes

- To make an attribute *private*, add a prefix __ to it
  - This makes the attribute accessible only to itself

- However, these are not foolproof…
  - A bad agent can always still access these attributes

How to: Attribute Protection

- Private attributes can still be accessed
- This can be avoided, but that is not for today
- Generally, python does not care (as much)
  About privacy as java does



```python
class A:
    def __init__(self):
        self.__name = "A"
        self.__value = 10

class B(A):
    def to_string(self):
        return f"{self.__name} {self.__value}"

b = B()

try:
    print("Option one")
    print(b.to_string())
    b.__name = "B"
    b.__value = 20
    print(b.to_string())
except:
    print("Nope! Option two")
    print(b._A__name, b._A__value)
    b._A__name = "B"
    b._A__value = 20
    print(b._A__name, b._A__value)
```

How to: Interfaces & Abstract Classes

- **Q:** Why do we use interfaces/abstract classes (in Java)?

How to: Interfaces & Abstract Classes

- **Q:** Why do we use interfaces/abstract classes (in Java)?
  - They provide a contract/guarantee for the compiler on what methods must exist
  - They help make the code more maintainable and readable

How to: Interfaces & Abstract Classes

- **Q:** Why do we use interfaces/abstract classes (in Java)?
  - They provide a contract/guarantee for the compiler on what methods must exist
  - They help make the code more maintainable and readable

- Python does (mostly) not care about type: so it does not care about interfaces

- The programmer should care!
  - How can we achieve what we need in python?

How to: Interfaces & Abstract Classes

- Python uses abstract base classes to raise exceptions when method flagged *@abstractmethod* are not implemented by a class inheriting it.

- Here, penguin1 is malformed as the abstract method is Left as such

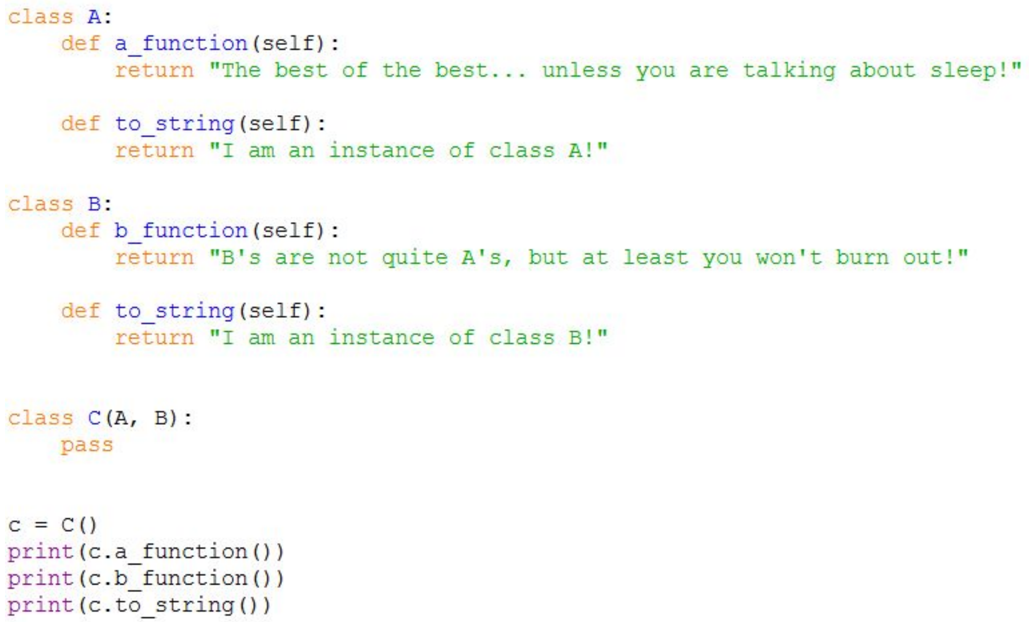
```
from abc import ABC, abstractmethod

class AbstractBird(ABC):
    def __init__(self, name):
        self.name = name
        super().__init__()

    @abstractmethod
    def can_fly(self):
        pass


class Penguin1 (AbstractBird):
    def __init__(self):
        super().__init__("Penguin")

class Penguin2 (AbstractBird):
    def __init__(self):
        super().__init__("Penguin")

    def can_fly(self):
        return False


try:
    penguin = Penguin1()
except:
    print("TypeError: Can't instantiate abstract
    class Penguin1 with abstract methods can_fly")

    penguin = Penguin2()

print(penguin.can_fly())
```

How to: Multiple Inheritance

- Python allows for multiple inheritance
  - Java only allows single inheritance (but multiple interfaces)

- However, mind your order!

- **Q:** What will the output of the code to the side be?



```python
class A:
    def a_function(self):
        return "The best of the best... unless you are talking about sleep!"

    def to_string(self):
        return "I am an instance of class A!"

class B:
    def b_function(self):
        return "B's are not quite A's, but at least you won't burn out!"

    def to_string(self):
        return "I am an instance of class B!"


class C(A, B):
    pass


c = C()
print(c.a_function())
print(c.b_function())
print(c.to_string())
```

How to: Multiple Inheritance

- Python allows for multiple inheritance
  - Java only allows single inheritance (but multiple interfaces)
- However, mind your order!
- **Q:** What will the output of the code to the side be?
- The **first** superclass is leading!

```python
class A:
    def a_function(self):
        return "The best of the best... unless you are talking about sleep!"

    def to_string(self):
        return "I am an instance of class A!"

class B:
    def b_function(self):
        return "B's are not quite A's, but at least you won't burn out!"

    def to_string(self):
        return "I am an instance of class B!"


class C(A, B):
    pass


c = C()
print(c.a_function())
print(c.b_function())
print(c.to_string())
```

How to: Strategy Pattern

- A strategy pattern is simply this:
  - Create a group of different classes that share crucial methods/attributes
- We will be using this later
- To do this: define (or imagine) an interface, and make sure each class has the important bits!

How to: Strategy Pattern

- A strategy pattern is simply this:
  - Create a group of different classes that share crucial methods/attributes
- We will be using this later
- To do this: define (or imagine) an interface, and make sure each class has the important bits!

How to: Decorator Pattern

- Goal: allow you to add extra values or functionality to an object, while maintaining what came before
- Each decoration only adds a small part to the whole, and relies on the previous ensemble to do the rest of the work

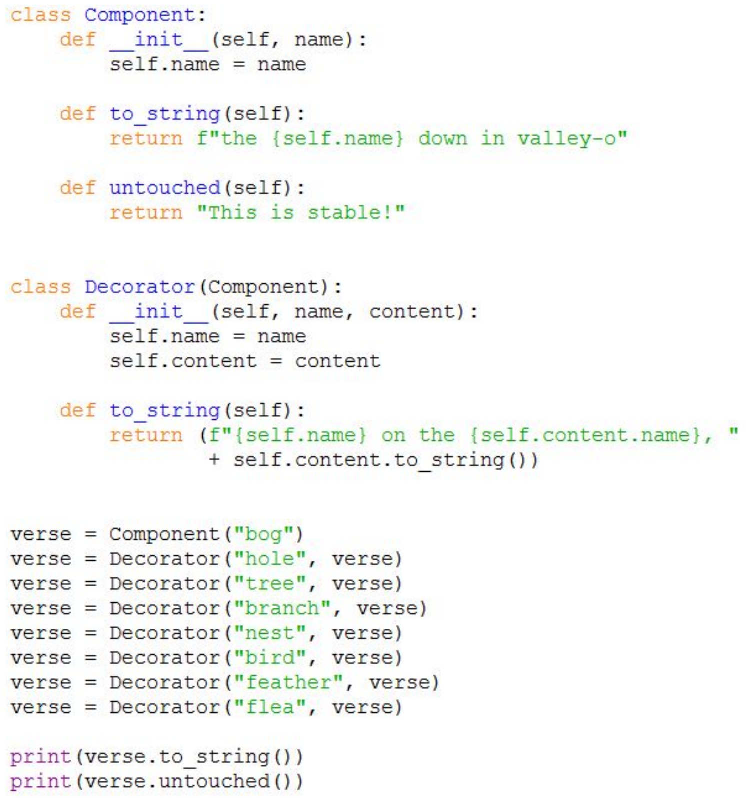
How to: Decorator Pattern

- Define a basic component
- Define one or multiple decorator classes that extend the methods of the component, and inherit the component class
- **Q:** What does this code output?

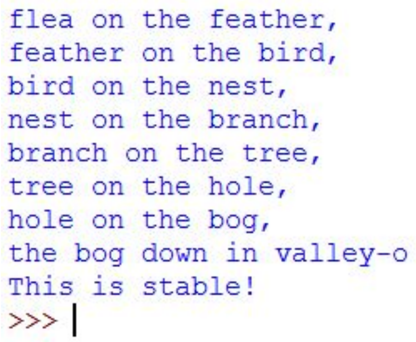
```python
class Component:
    def __init__(self, name):
        self.name = name

    def to_string(self):
        return f"the {self.name} down in valley-o"

    def untouched(self):
        return "This is stable!"


class Decorator(Component):
    def __init__(self, name, content):
        self.name = name
        self.content = content

    def to_string(self):
        return (f"{self.name} on the {self.content.name}, "
                + self.content.to_string())


verse = Component("bog")
verse = Decorator("hole", verse)
verse = Decorator("tree", verse)
verse = Decorator("branch", verse)
verse = Decorator("nest", verse)
verse = Decorator("bird", verse)
verse = Decorator("feather", verse)
verse = Decorator("flea", verse)

print(verse.to_string())
print(verse.untouched())
```

How to: Decorator Pattern

- Define a basic component
- Define one or multiple decorator classes that extend the methods of the component, and inherit the component class
- **Q:** What does this code output?



```python
class Component:
    def __init__(self, name):
        self.name = name

    def to_string(self):
        return f"the {self.name} down in valley-o"

    def untouched(self):
        return "This is stable!"


class Decorator(Component):
    def __init__(self, name, content):
        self.name = name
        self.content = content

    def to_string(self):
        return (f"{self.name} on the {self.content.name}, "
                + self.content.to_string())


verse = Component("bog")
verse = Decorator("hole", verse)
verse = Decorator("tree", verse)
verse = Decorator("branch", verse)
verse = Decorator("nest", verse)
verse = Decorator("bird", verse)
verse = Decorator("feather", verse)
verse = Decorator("flea", verse)

print(verse.to_string())
print(verse.untouched())
```



```
flea on the feather,
feather on the bird,
bird on the nest,
nest on the branch,
branch on the tree,
tree on the hole,
hole on the bog,
the bog down in valley-o
This is stable!
>>> |
```

How to: Iterator Pattern

- Goal: allow us to grab elements or properties one by one until we've gone through them all (or met some other condition)

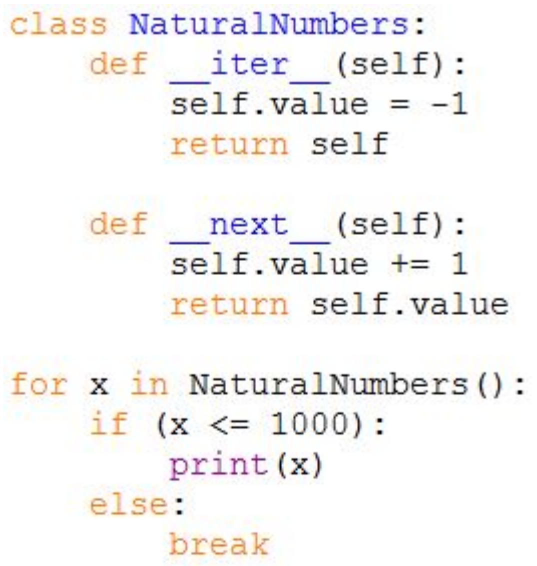
- In python: **for *element* in *iterable*:**

- Most for-loops in python are actually iterators
  - In addition, all lists are iterable

- In a class: __*iter*__ and __*next*__ are the keywords

How to: Iterator Pattern

- __iter__ defines the state of the class at the start of iteration.
- __next__ defines what happens each time the iterator requests the next element, including the first time! (hence the -1 instead of 0 at the start)
- This example will keep on increasing the value by 1 until overflow

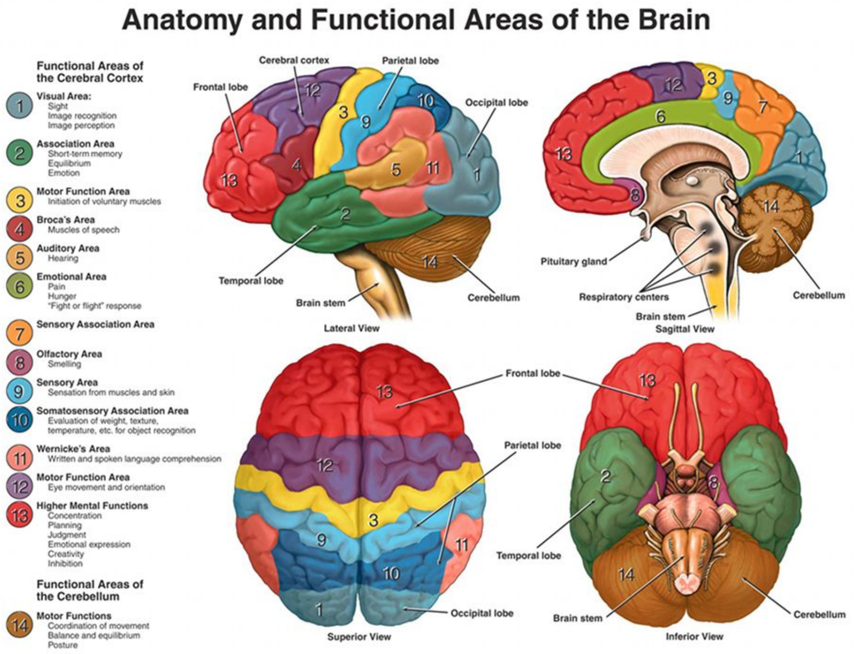


```
class NaturalNumbers:
    def __iter__(self):
        self.value = -1
        return self

    def __next__(self):
        self.value += 1
        return self.value

for x in NaturalNumbers():
    if (x <= 1000):
        print(x)
    else:
        break
```

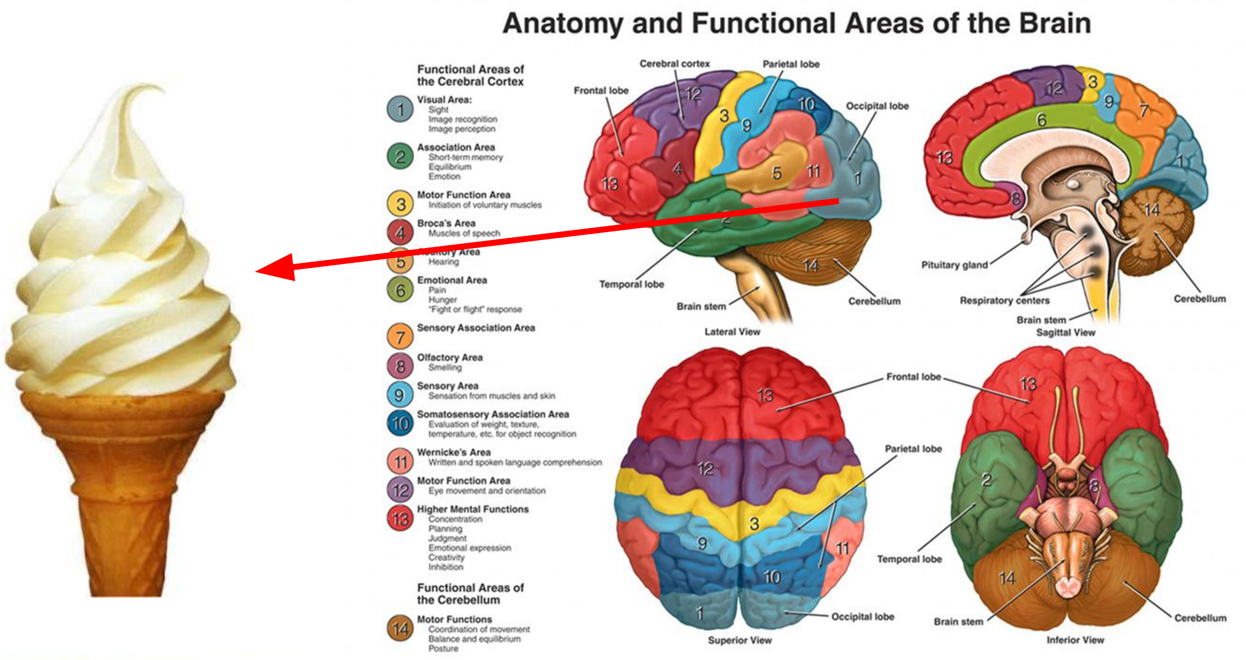
How to: MVC Pattern

- Goal: Structure a UI program such that flow of information and responsibilities are kept separate and structured
- **M**odel: keeps track of the program’s data structure, logic and rules
- **V**iew: allows to user to view the data represented by the model
- **C**ontroller: responds to user input and changes data (after validation)

How to: MVC Pattern


Anatomy and Functional Areas of the Brain
Functional Areas of the Cerebral Cortex
1 Visual Area: Sight, Image recognition, Image perception
2 Association Area: Short-term memory, Equilibrium, Emotion
3 Motor Function Area: Initiation of voluntary muscles
4 Broca's Area: Muscles of speech
5 Auditory Area: Hearing
6 Emotional Area: Pain, Hunger, "Fight or flight" response
7 Sensory Association Area
8 Olfactory Area: Smelling
9 Sensory Area: Sensation from muscles and skin
10 Somatosensory Association Area: Evaluation of weight, texture, temperature, etc. for object recognition
11 Wernicke's Area: Written and spoken language comprehension
12 Motor Function Area: Eye movement and orientation
13 Higher Mental Functions: Concentration, Planning, Judgment, Emotional expression, Creativity, Inhibition
Functional Areas of the Cerebellum
14 Motor Functions: Coordination of movement, Balance and equilibrium, Posture
Cerebral cortex
Parietal lobe
Frontal lobe
Occipital lobe
Temporal lobe
Brain stem
Cerebellum
Lateral View
Pituitary gland
Respiratory centers
Brain stem
Cerebellum
Sagittal View
Frontal lobe
Parietal lobe
Temporal lobe
Occipital lobe
Superior View
Brain stem
Cerebellum
Inferior View

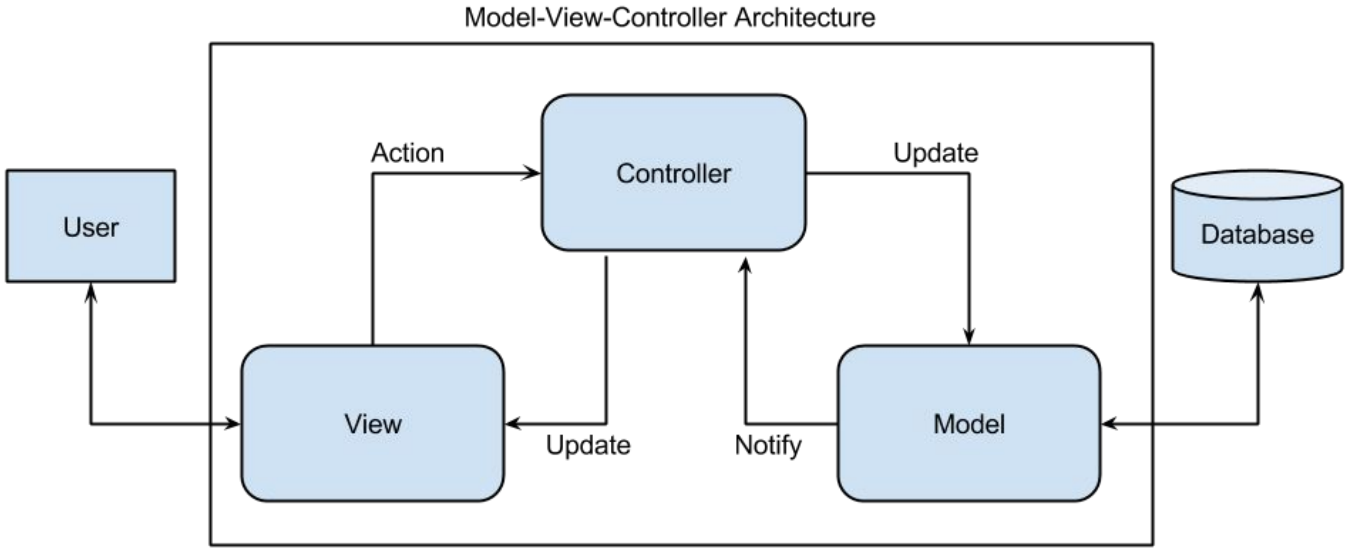
How to: MVC Pattern



How to: MVC Pattern

- Often surprisingly complex
- It is easy, and sometimes necessary to overlap view and controller
- In principle, the order of how to implement MVC is as follows:
  - The model is made and needs to know nothing
  - The controller needs access to the model and view
  - The view needs access to the model, as well as the controller to alert it of user input
- Alternatively, the controller flows information from view to model and back
- Often multiple components require ‘sight’ of each other

How to: MVC Pattern



Thank you for listening!