Week 11: Classes and Object-oriented Programming

POP77001 Computer Programming for Social Scientists

Tom Paskhalis

21 November 2022

Module website: tinyurl.com/POP77001

Overview

- Abstraction and decomposition
- Object attributes
- Object-oriented programming (OOP)
- Classes
- Methods
- Class inheritance

Recap: Python conceptual hierarchy

Python programs can be decomposed into modules, statements, expressions, and objects, as follows:

1. **Programs** are composed of **modules**.
2. **Modules** contain **statements**.
3. **Statements** contain **expressions**
4. **Expressions** create and process **objects**.

Recap: Python objects

- Everything that Python operates on is an **object**.
- This includes numbers, strings, data structures, functions, etc.
- Eact object has a **type** (e.g. string or function) and internal **data**.
- Objects can be **mutable** (e.g. list) and **immutable** (e.g. string).

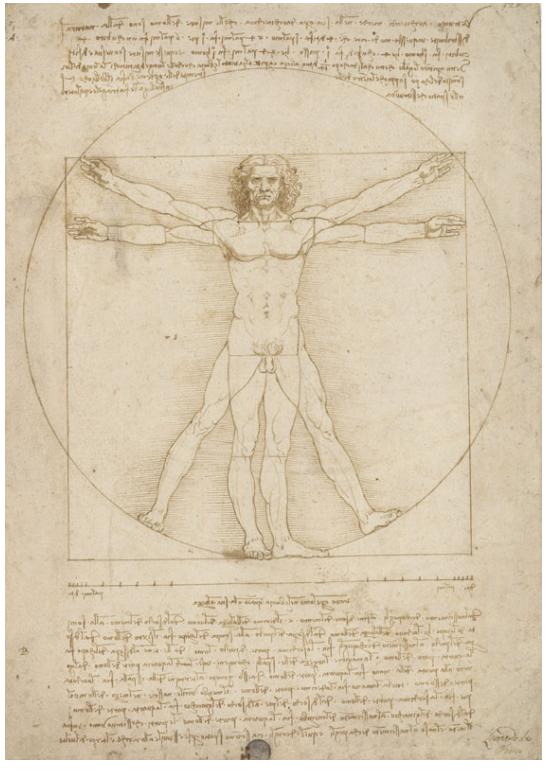
Recap: Decomposition and Abstraction

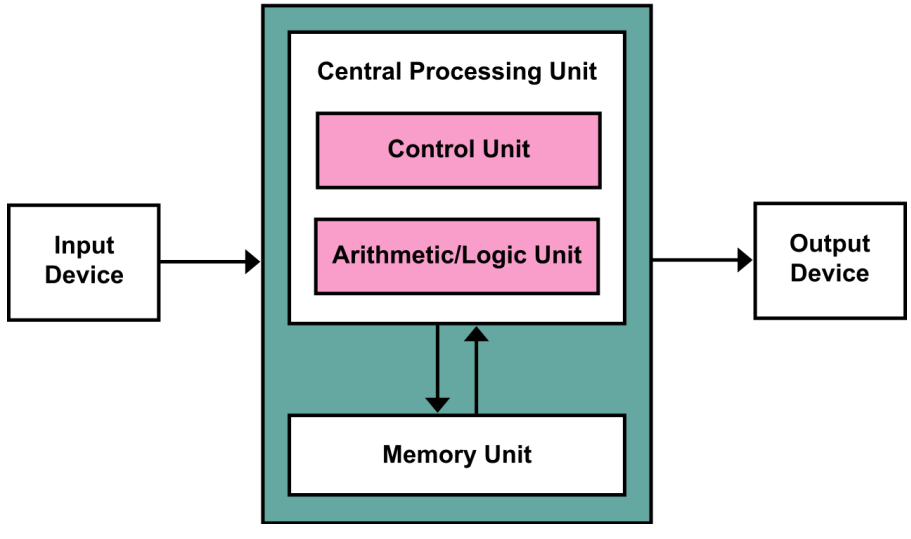
Source: IKEA

Achieving Decomposition and Abstraction

- So far: modules, functions.
- But modules and functions only abstract **code**.
- Not **data**.
- Hence, we need something else.
- **Classes**!

Abstraction





Source: Gallerie dell'Accademia, Wikipedia

Python objects we have seen so far

- Built-in types (integers, strings, lists, etc.)
- Imported from external packages (arrays, data frames, etc.)

Python objects we have seen so far

- Built-in types (integers, strings, lists, etc.)
- Imported from external packages (arrays, data frames, etc.)

In [2]:
```
s = 'watermelon'
sr = pd.Series([7, 1, 19])
```

Python objects we have seen so far

- Built-in types (integers, strings, lists, etc.)
- Imported from external packages (arrays, data frames, etc.)

In [2]:
```
s = 'watermelon'
sr = pd.Series([7, 1, 19])
```

Note the syntactic similarity between the two lines below:

Python objects we have seen so far

- Built-in types (integers, strings, lists, etc.)
- Imported from external packages (arrays, data frames, etc.)

In [2]:
```
s = 'watermelon'
sr = pd.Series([7, 1, 19])
```

Note the syntactic similarity between the two lines below:

In [3]:
```
s.upper
```

Out[3]:
```
<function str.upper()>
```

Python objects we have seen so far

- Built-in types (integers, strings, lists, etc.)
- Imported from external packages (arrays, data frames, etc.)

In [2]:
```
s = 'watermelon'
sr = pd.Series([7, 1, 19])
```

Note the syntactic similarity between the two lines below:

In [3]:
```
s.upper
```

Out[3]:
```
<function str.upper()>
```

In [4]:
```
sr.shape
```

Out[4]:
```
(3,)
```

Python object attributes

- Attributes are objects that are associated with a specific type.
- The constitute the essence of object-oriented programming in Python.

```
object.attribute
```

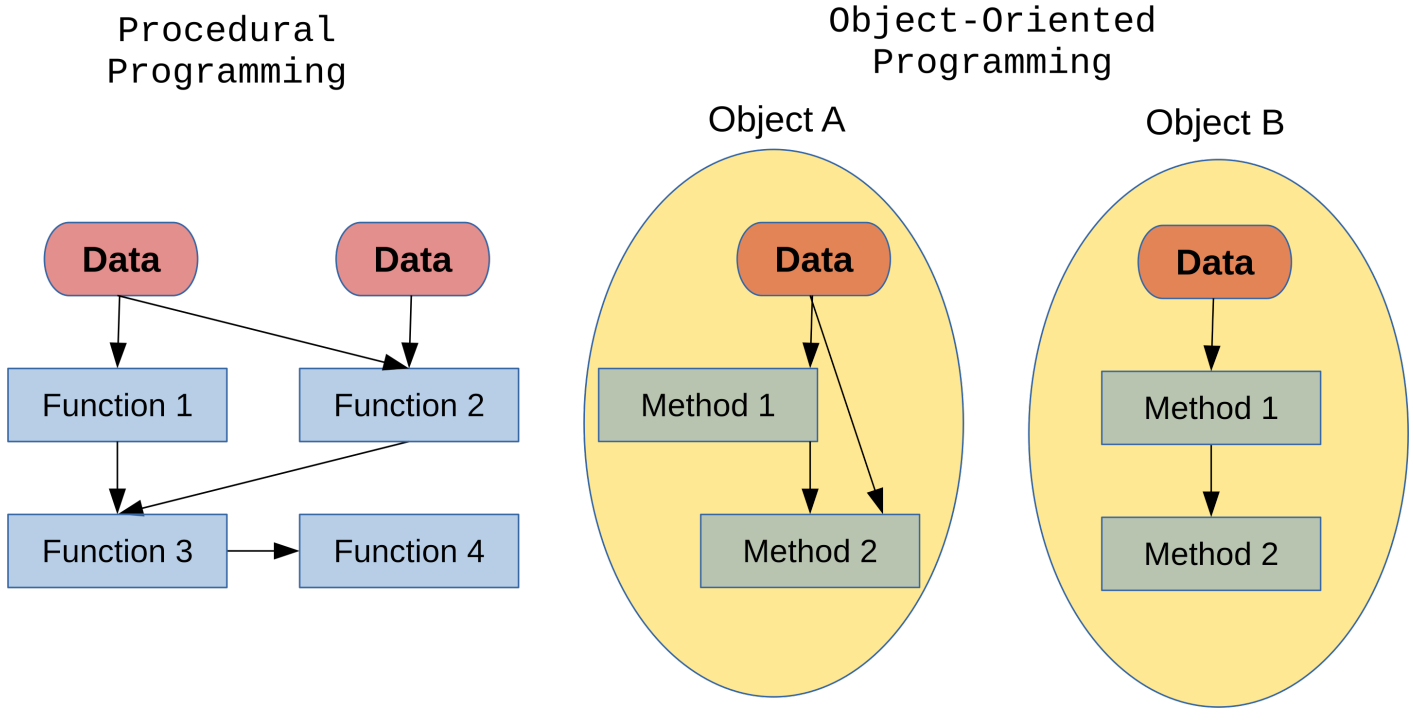
- This expression effectively means:

> *Find the first occurrence of attribute by looking in `object`, then in all classes above it.*

Object-based vs Object-oriented programming

- Until now our code was **object-based**.
- We created and passed objects around our programs.
- For our code to be truly **object-oriented**,
- Our objects need to be part of **inheritance hierarchy**.

Procedural vs Object-oriented programming



Source: Wellcome Collection
Extra: The Rise of Virtual Pets

Class definition

Class definition

In [5]:

```
from datetime import date

class Tamagotchi(object):
    """Creates a new Tamagotchi and gives it a name"""
    def __init__(self, name, birthdate = date.today()):
        self.name = name
        self.birthdate = birthdate
        self.food = 0
    def get_age(self):
        """Get Tamagotchi's name in days"""
        return date.today() - self.birthdate
    def feed(self):
        """Give Tamagotchi some food"""
        self.food += 1
    def play(self):
        """Play with Tamagotchi"""
        self.food -= 1
    def __str__(self):
        return (self.name + ' - ' +
                'Age: ' + str(self.get_age().days) + ' days ' +
                'Food: ' + str(self.food))
```

Class definition explained

- Class: `Tamagotchi`
- Data attributes:
    - `name` - name given as string
    - `birthdate` - birth date expressed as `datetime.date`
    - `food` - food level expressed as integer
- Methods (functions attached to this class):
    - `__init__()` - constructor, called when an object of this class is created.
    - `__str__()` - called when an object of this class is printed (with `print()` or `str()`)
    - `get_age()` - retrieve age expressed as `datetime.timedelta`
    - `feed()` - increment food level by 1
    - `play()` - decrement food level by 1

Class instantiation

Class instantiation

In [6]:
```
kuchipatchi = Tamagotchi("Kuchipatchi", date(2022, 5, 18))
```

Class instantiation

In [6]:
```
kuchipatchi = Tamagotchi("Kuchipatchi", date(2022, 5, 18))
```

In [7]:
```
type(kuchipatchi) # Check object type
```

Out[7]:
```
__main__.Tamagotchi
```

Class instantiation

In [6]:
```
kuchipatchi = Tamagotchi("Kuchipatchi", date(2022, 5, 18))
```

In [7]:
```
type(kuchipatchi) # Check object type
```

Out[7]:
```
__main__.Tamagotchi
```

In [8]:
```
kuchipatchi.name # Access object's data attribute
```

Out[8]:
```
'Kuchipatchi'
```

Class instantiation

In [6]:
```
kuchipatchi = Tamagotchi("Kuchipatchi", date(2022, 5, 18))
```

In [7]:
```
type(kuchipatchi) # Check object type
```

Out[7]:
```
__main__.Tamagotchi
```

In [8]:
```
kuchipatchi.name # Access object's data attribute
```

Out[8]:
```
'Kuchipatchi'
```

In [9]:
```
kuchipatchi.feed() # Invoke object method
```

Class instantiation

In [6]:
```
kuchipatchi = Tamagotchi("Kuchipatchi", date(2022, 5, 18))
```

In [7]:
```
type(kuchipatchi) # Check object type
```

Out[7]:
```
__main__.Tamagotchi
```

In [8]:
```
kuchipatchi.name # Access object's data attribute
```

Out[8]:
```
'Kuchipatchi'
```

In [9]:
```
kuchipatchi.feed() # Invoke object method
```

In [10]:
```
print(kuchipatchi)
```

```
Kuchipatchi - Age: 187 days Food: 1
```

What is class?

- Classes are factories for generating one or more objects of the same type.
- Every time we call (instantiate) a class we create a new object (instance) with distinct namespace.

What is class?

- Classes are factories for generating one or more objects of the same type.
- Every time we call (instantiate) a class we create a new object (instance) with distinct namespace.

In [11]:
```
mimitchi = Tamagotchi("Mimitchi", date(2022, 8, 8))
print(type(mimitchi))
print(mimitchi)
```

```
<class '__main__.Tamagotchi'>
Mimitchi - Age: 105 days Food: 0
```

What is class?

- Classes are factories for generating one or more objects of the same type.
- Every time we call (instantiate) a class we create a new object (instance) with distinct namespace.

In [11]:
```
mimitchi = Tamagotchi("Mimitchi", date(2022, 8, 8))
print(type(mimitchi))
print(mimitchi)
```

```
<class '__main__.Tamagotchi'>
Mimitchi - Age: 105 days Food: 0
```

In [12]:
```
sebiretchi = Tamagotchi("Sebiretchi", date(2022, 11, 1))
print(type(sebiretchi))
print(sebiretchi)
```

```
<class '__main__.Tamagotchi'>
Sebiretchi - Age: 20 days Food: 0
```

Classes vs Objects

- In our example above `Tamagotchi` is a class.
- `Kuchipatchi`, `Mimitchi`, `Sebiretchi` are instances of the class `Tamagotchi`.
- In other words, they are objects of type `Tamagotchi`.
- The same way as `str` is a class and `'watermelon'` is an object of type `str`.

Class methods

- Functions associated with a specific class are called **methods**.
- These functions are simultaneously class attributes.
- Hence, their syntax is `object.method()` as opposed to `function(object)`.

Class methods

- Functions associated with a specific class are called **methods**.
- These functions are simultaneously class attributes.
- Hence, their syntax is `object.method()` as opposed to `function(object)`.

In [13]:
```
print(kuchipatchi)
print(mimitchi)
```

```
Kuchipatchi - Age: 187 days Food: 1
Mimitchi - Age: 105 days Food: 0
```

Class methods

- Functions associated with a specific class are called **methods**.
- These functions are simultaneously class attributes.
- Hence, their syntax is `object.method()` as opposed to `function(object)`.

In [13]:
```
print(kuchipatchi)
print(mimitchi)
```

```
Kuchipatchi - Age: 187 days Food: 1
Mimitchi - Age: 105 days Food: 0
```

In [14]:
```
kuchipatchi.feed() # Invoke object method
```

Class methods

- Functions associated with a specific class are called **methods**.
- These functions are simultaneously class attributes.
- Hence, their syntax is `object.method()` as opposed to `function(object)`.

In [13]:
```
print(kuchipatchi)
print(mimitchi)
```

```
Kuchipatchi - Age: 187 days Food: 1
Mimitchi - Age: 105 days Food: 0
```

In [14]:
```
kuchipatchi.feed() # Invoke object method
```

In [15]:
```
# Methods modify only the data attritubes
# of the associated object
print(kuchipatchi)
print(mimitchi)
```

```
Kuchipatchi - Age: 187 days Food: 2
Mimitchi - Age: 105 days Food: 0
```

Special methods

- Some methods start and end with double underscore (`__`).
- These methods serve special purposes.
- Usually, they are not expected to be invoked directly.
- Examples of special methods:
    - `__init__()` - defines object instantiation;
    - `__str__()` - defines how an object is printed out;
    - `__add__()` - overloads the `+` operator
        - also `__sub__()` for `-`, `__mul__()` for `*`, etc.
    - `__eq__()` - overloads the `==` operator
        - also `__lt__()` for `<`, `__ge__()` for `>=`, etc.
    - `__len__()` - returns the length of the object (is called by `len()` function)
    - `__iter__()` - returns an iterator (used in loops)

Extra: Special method names

`self`

- Variable that references the current instance of the class.
- The name is a convention, but a strong one.

```
def __init__(self, name):
    self.name = name
```

Overloading

- Classes may override most built-in operators
- E.g. function `sorted()` requires `__lt__()` (less than) method to be implemented.

Overloading

- Classes may override most built-in operators
- E.g. function `sorted()` requires `__lt__()` (less than) method to be implemented.

In [16]:

```
class Tamagotchi(object):
    """Creates a new Tamagotchi and gives it a name"""
    def __init__(self, name, birthdate = date.today()):
        self.name = name
        self.birthdate = birthdate
        self.food = 0
    def get_age(self):
        """Get Tamagotchi's name in days"""
        return date.today() - self.birthdate
    def feed(self):
        """Give Tamagotchi some food"""
        self.food += 1
    def play(self):
        """Play with Tamagotchi"""
        self.food -= 1
    def __lt__(self, other):
        """Returns True if self's name precedes other's name alphabetic
        return self.name < other.name
    def __str__(self):
        return (self.name + ' - ' +
```

Overloading example

- Since we changes class definition above we need to recreate objects to change their behaviour.

Overloading example

- Since we changes class definition above we need to recreate objects to change their behaviour.

In [17]:
```
mimitchi = Tamagotchi("Mimitchi", date(2022, 8, 8))
```

Overloading example

- Since we changes class definition above we need to recreate objects to change their behaviour.

In [17]:
```
mimitchi = Tamagotchi("Mimitchi", date(2022, 8, 8))
```

In [18]:
```
sebiretchi = Tamagotchi("Sebiretchi", date(2022, 11, 1))
```

Overloading example

- Since we changes class definition above we need to recreate objects to change their behaviour.

In [17]:
```
mimitchi = Tamagotchi("Mimitchi", date(2022, 8, 8))
```

In [18]:
```
sebiretchi = Tamagotchi("Sebiretchi", date(2022, 11, 1))
```

In [19]:
```
mimitchi < sebiretchi
```

Out[19]:
```
True
```

Overloading example continued

Overloading example continued

In [20]:
```
sorted([mimitchi, sebiretchi])
```

Out[20]:
```
[<__main__.Tamagotchi at 0x7f8eaa6c8730>,
 <__main__.Tamagotchi at 0x7f8e566633a0>]
```

Overloading example continued

In [20]:
```
sorted([mimitchi, sebiretchi])
```

Out[20]:
```
[<__main__.Tamagotchi at 0x7f8eaa6c8730>,
 <__main__.Tamagotchi at 0x7f8e566633a0>]
```

In [21]:
```
print([str(x) for x in sorted([mimitchi, sebiretchi])])
```

```
['Mimitchi - Age: 105 days Food: 0', 'Sebiretchi - Age: 20 days
Food: 0']
```

Inheritance

- Classes allows customization by **inheritance**.
- New components can be introduced in **subclasses**.
- Without having to re-implement functionality from scratch,
- Classes can inherit attributes from **superclasses**.
- This can create a hierarchy of classes,
- At the top of which is class `object`.

Superclass

Superclass

In [22]:
```
class Survey(object):
    """Creates a new Survey"""
    def __init__(self):
        """Initialize a new Survey with an empty questionnaire"""
        self.questionnaire = []
    def add_question(self, question):
        """Add question to the questionnaire"""
        self.questionnaire.append(question)
    def __add__(self, other):
        """Combine Surveys together"""
        return self.questionnaire + other.questionnaire
    def __len__(self):
        """Returns the length of Survey questionnaire"""
        return len(self.questionnaire)
```

Superclass

In [22]:
```
class Survey(object):
    """Creates a new Survey"""
    def __init__(self):
        """Initialize a new Survey with an empty questionnaire"""
        self.questionnaire = []
    def add_question(self, question):
        """Add question to the questionnaire"""
        self.questionnaire.append(question)
    def __add__(self, other):
        """Combine Surveys together"""
        return self.questionnaire + other.questionnaire
    def __len__(self):
        """Returns the length of Survey questionnaire"""
        return len(self.questionnaire)
```

In [23]:
```
survey1 = Survey()
```

Superclass

In [22]:
```
class Survey(object):
    """Creates a new Survey"""
    def __init__(self):
        """Initialize a new Survey with an empty questionnaire"""
        self.questionnaire = []
    def add_question(self, question):
        """Add question to the questionnaire"""
        self.questionnaire.append(question)
    def __add__(self, other):
        """Combine Surveys together"""
        return self.questionnaire + other.questionnaire
    def __len__(self):
        """Returns the length of Survey questionnaire"""
        return len(self.questionnaire)
```

In [23]:
```
survey1 = Survey()
```

In [24]:
```
survey1.add_question("What is your age?")
```

Superclass

In [22]:
```
class Survey(object):
    """Creates a new Survey"""
    def __init__(self):
        """Initialize a new Survey with an empty questionnaire"""
        self.questionnaire = []
    def add_question(self, question):
        """Add question to the questionnaire"""
        self.questionnaire.append(question)
    def __add__(self, other):
        """Combine Surveys together"""
        return self.questionnaire + other.questionnaire
    def __len__(self):
        """Returns the length of Survey questionnaire"""
        return len(self.questionnaire)
```

In [23]:
```
survey1 = Survey()
```

In [24]:
```
survey1.add_question("What is your age?")
```

In [25]:
```
len(survey1)
```

Out[25]: 1

Subclass

Subclass

In [26]:

```
class TelephoneSurvey(Survey):

    # This is a class variable
    # When a new instance of class is created
    # a new instance of next_survey_id is not
    next_survey_id = 1

    def __init__(self):
        super().__init__()
        self.survey_id = TelephoneSurvey.next_survey_id
        TelephoneSurvey.next_survey_id += 1
        self.mode = 'Telephone'
```

Subclass

In [26]:
```
class TelephoneSurvey(Survey):

    # This is a class variable
    # When a new instance of class is created
    # a new instance of next_survey_id is not
    next_survey_id = 1

    def __init__(self):
        super().__init__()
        self.survey_id = TelephoneSurvey.next_survey_id
        TelephoneSurvey.next_survey_id += 1
        self.mode = 'Telephone'
```

In [27]:
```
survey2 = TelephoneSurvey()
```

Subclass

In [26]:
```
class TelephoneSurvey(Survey):

    # This is a class variable
    # When a new instance of class is created
    # a new instance of next_survey_id is not
    next_survey_id = 1

    def __init__(self):
        super().__init__()
        self.survey_id = TelephoneSurvey.next_survey_id
        TelephoneSurvey.next_survey_id += 1
        self.mode = 'Telephone'
```

In [27]:
```
survey2 = TelephoneSurvey()
```

In [28]:
```
survey2.add_question("What is your age?")
```

Subclass

In [26]:
```
class TelephoneSurvey(Survey):

    # This is a class variable
    # When a new instance of class is created
    # a new instance of next_survey_id is not
    next_survey_id = 1

    def __init__(self):
        super().__init__()
        self.survey_id = TelephoneSurvey.next_survey_id
        TelephoneSurvey.next_survey_id += 1
        self.mode = 'Telephone'
```

In [27]:
```
survey2 = TelephoneSurvey()
```

In [28]:
```
survey2.add_question("What is your age?")
```

In [29]:
```
len(survey2)
```

Out[29]:
```
1
```

Inheritance Hierarchy

Inheritance Hierarchy

In [30]:
```
class Person(object):
    pass
```

Inheritance Hierarchy

In [30]:
```
class Person(object):
    pass
```

In [31]:
```
class Employee(Person):
    pass
```

Inheritance Hierarchy

In [30]:
```
class Person(object):
    pass
```

In [31]:
```
class Employee(Person):
    pass
```

In [32]:
```
class Admin(Employee):
    pass
```

Inheritance Hierarchy

In [30]:
```
class Person(object):
    pass
```

In [31]:
```
class Employee(Person):
    pass
```

In [32]:
```
class Admin(Employee):
    pass
```

In [33]:
```
class Academic(Employee):
    pass
```

Inheritance Hierarchy continued

Inheritance Hierarchy continued

In [34]:
```
prof = Academic()
```

Inheritance Hierarchy continued

In [34]:
```
prof = Academic()
```

In [35]:
```
isinstance(prof, Person)
```

Out[35]:
```
True
```

Inheritance Hierarchy continued

In [34]:
```
prof = Academic()
```

In [35]:
```
isinstance(prof, Person)
```

Out[35]: True

In [36]:
```
isinstance(prof, Employee)
```

Out[36]: True

Inheritance Hierarchy continued

In [34]:
```
prof = Academic()
```

In [35]:
```
isinstance(prof, Person)
```

Out[35]:
```
True
```

In [36]:
```
isinstance(prof, Employee)
```

Out[36]:
```
True
```

In [37]:
```
isinstance(prof, Admin)
```

Out[37]:
```
False
```

Classes and OOP in Python

- Classes are the core of OOP.
- Classes bundle data with functions.
- They allow for objects to be part of inheritance hierarchy.
- In general, OOP in Python is entirely optional.
- For some tasks the level of abstraction provided by functions and modules is sufficient.
- But for some applications (user-facing, large projects, high-reliability) OOP is essential.

Next

- Tutorial: Python objects, classes and methods
- Assignment 4: Due at 12:00 on Monday, 28th November (submission on Blackboard)
- Next week: Complexity and performance