Week 11: Classes and Object-oriented Programming

POP77001 Computer Programming for Social Scientists

Tom Paskhalis

Overview

- Decomposition and abstraction
- Object attributes
- Object-oriented programming (OOP)
- Classes
- Methods
- Class inheritance

Decomposition and Abstraction

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**.

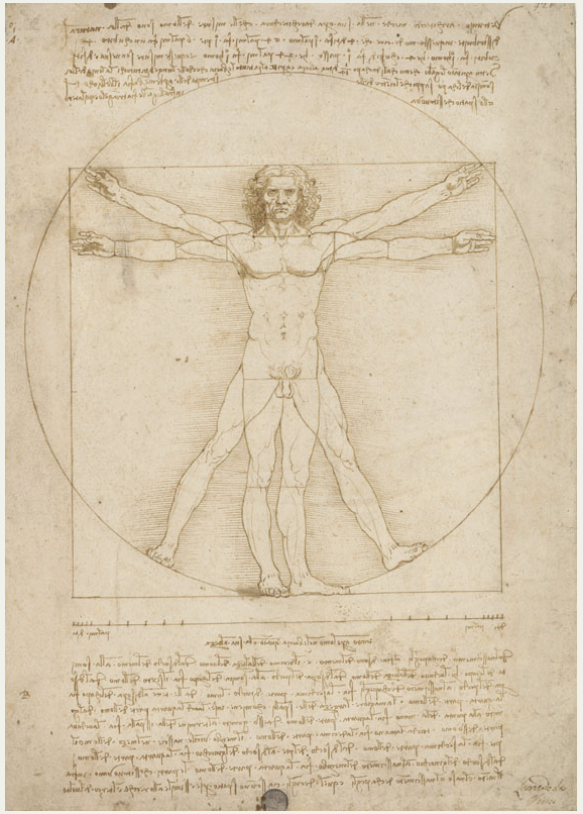
Recap: Python Objects

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

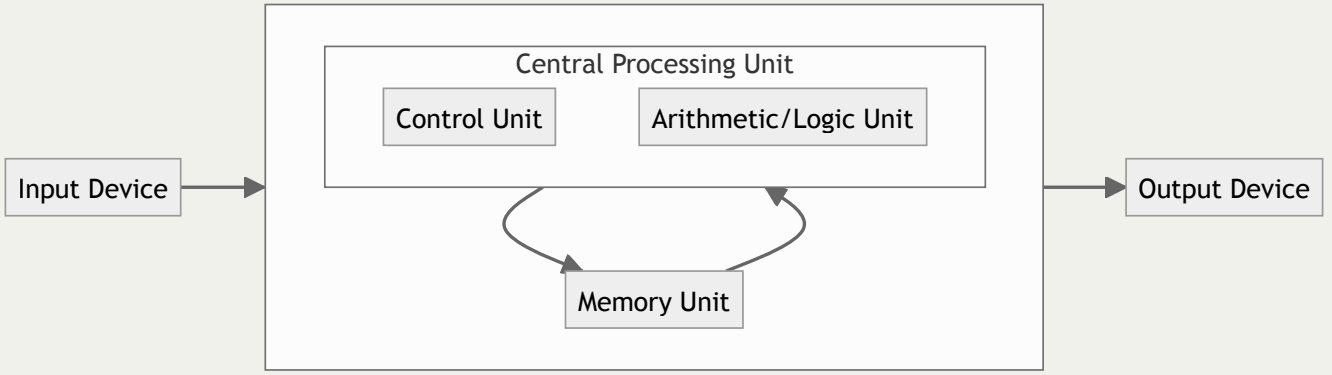
Achieving Decomposition and Abstraction

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

Abstraction



(Gallerie dell'Accademia)



$$y = X\beta + \epsilon$$

OOP

Python Objects So Far

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

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

Note the syntactic similarity between the two lines below:

```
1 s.upper
```

```
<built-in method upper of str object at 0x78d8a25f3970>
```

```
1 ser.shape
```

```
(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:

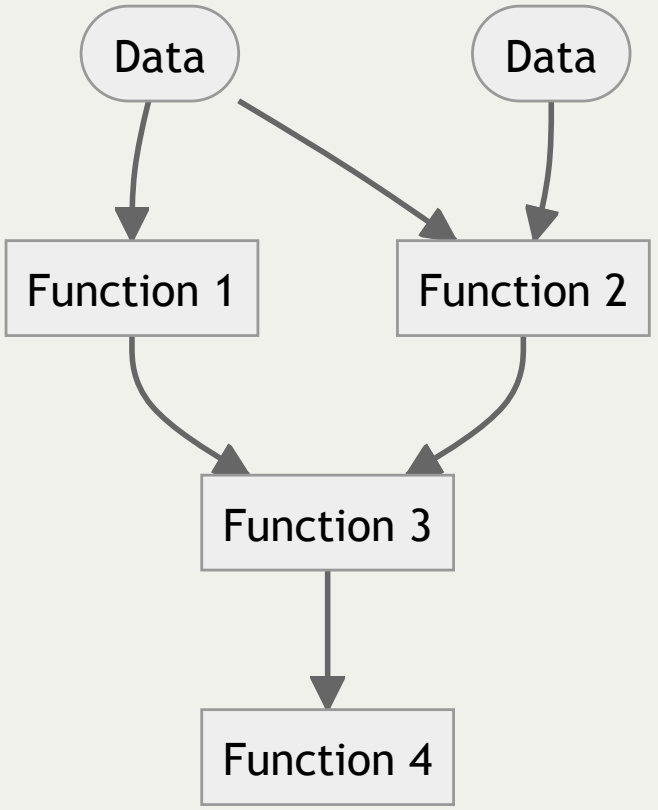
> Find the first occurrence of attribute by looking in `object`, then in all classes above it.

Object-based vs Object-oriented Programming

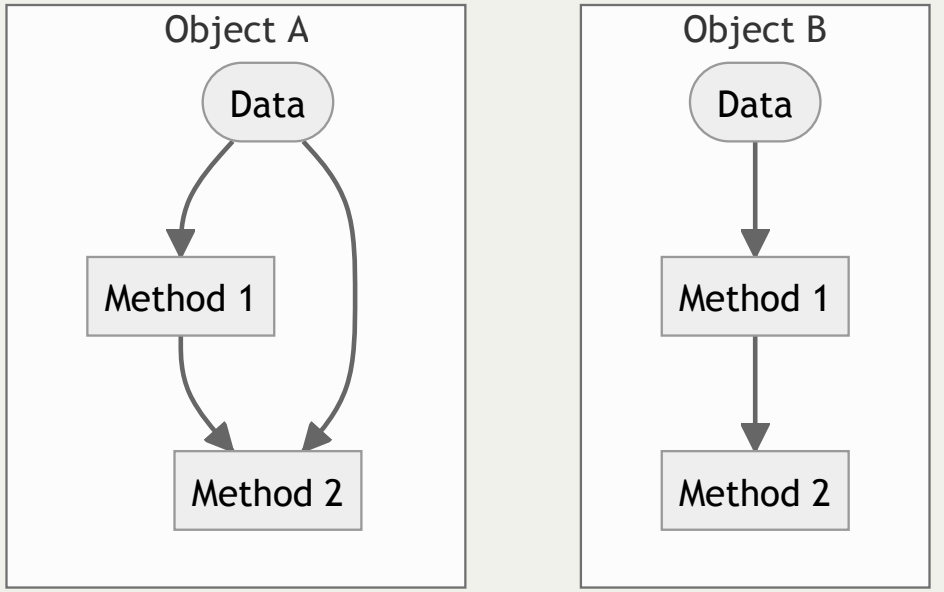
- Until now our code was **object-based**.
- We created and passed objects around our programs.
- For our code to be truly **object-oriented**,
- Our objects need to be part of **inheritance hierarchy**.

Procedural vs Object-oriented Programming

Procedural Programming



Object-oriented Programming



Classes

(Wellcome Collection)

Extra

The Rise of Virtual Pets

Class Definition

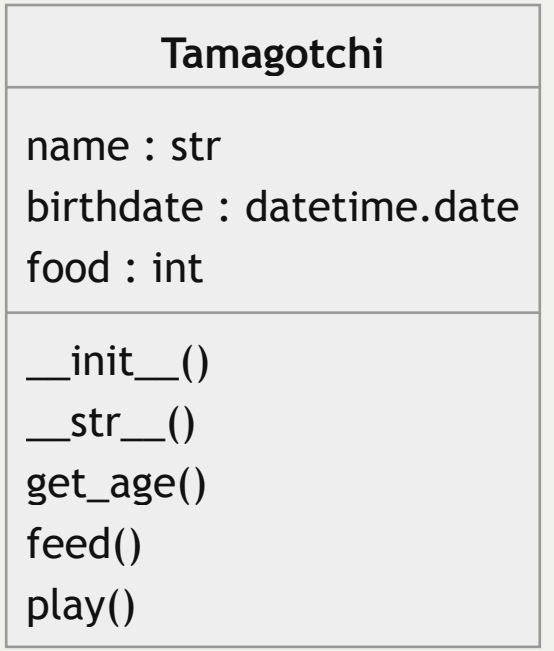
```
1  from datetime import date
2
3  class Tamagotchi:
4      """Class modelling a simple Tamagotchi toy"""
5      def __init__(self, name, birthdate = date.today()):
6          """Creates a new Tamagotchi and gives it a name"""
7          self.name = name
8          self.birthdate = birthdate
9          self.food = 0
10     def __str__(self):
11         """Returns a string representation of Tamagotchi"""
12         return (self.name + ' - ' +
13                 'Age: ' + str(self.get_age().days) + ' days ' +
14                 'Food: ' + str(self.food))
15     def get_age(self):
16         """Get Tamagotchi's name in days"""
17         return date.today() - self.birthdate
18     def feed(self):
19         """Give Tamagotchi some food"""
20         self.food += 1
21     def play(self):
22         """Play with Tamagotchi"""
23         self.food -= 1
```

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 Diagram

- Class diagrams are a common way of graphically representing classes and their relations.
- UML (Unified Modeling Language) provides a standard notation for class diagrams.
- 3 components of a class diagram in UML-style include:
  - Class name (top compartment)
  - Data attributes (middle compartment)
  - Methods (bottom compartment)



Class Instantiation

```
1 kuchipatchi = Tamagotchi("Kuchipatchi", date(2023, 5, 20))
```

```
1 type(kuchipatchi) # Check object type
```

```
<class '__main__.Tamagotchi'>
```

```
1 kuchipatchi.name # Access object's data attribute
```

```
'Kuchipatchi'
```

```
1 kuchipatchi.feed() # Invoke object method
```

```
1 print(kuchipatchi)
```

```
Kuchipatchi - Age: 548 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.

```
1 mimitchi = Tamagotchi("Mimitchi", date(2023, 8, 8))
2 type(mimitchi)
```

```
<class '__main__.Tamagotchi'>
```

```
1 print(mimitchi)
```

```
Mimitchi - Age: 468 days Food: 0
```

```
1 sebiretchi = Tamagotchi("Sebiretchi", date(2023, 11, 1))
2 type(sebiretchi)
```

```
<class '__main__.Tamagotchi'>
```

```
1 print(sebiretchi)
```

```
Sebiretchi - Age: 383 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`.

Methods

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)`.

```
1 print(kuchipatchi)
```

```
Kuchipatchi - Age: 548 days Food: 1
```

```
1 print(mimitchi)
```

```
Mimitchi - Age: 468 days Food: 0
```

```
1 kuchipatchi.feed() # Invoke object method
```

```
1 # Methods modify only the data attributes
2 # of the associated object
3 print(kuchipatchi)
```

```
Kuchipatchi - Age: 548 days Food: 2
```

```
1 print(mimitchi)
```

```
Mimitchi - Age: 468 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
```

```
1  def __init__(self, name, birthdate = date.today()):
2      """Creates a new Tamagotchi and gives it a name"""
3      self.name = name
4      self.birthdate = birthdate
5      self.food = 0
```

Polymorphism

- **Polymorphism** is one of the most powerful concepts in OOP.
- It allows to use the same interface for objects of different classes.
- The dynamic nature of Python makes it possible to use polymorphism even without inheritance.
- E.g., `sorted()` function can sort objects of different types.
- It is possible for all objects that have `__lt__()` (**l**ess **t**han) method.

Polymorphism: Example

```
class Tamagotchi:
    """Class modelling a simple Tamagotchi toy"""
    def __init__(self, name, birthdate = date.today()):
        """Creates a new Tamagotchi and gives it a name"""
        self.name = name
        self.birthdate = birthdate
        self.food = 0
    def __lt__(self, other):
        """Returns True if self's name precedes other's name alphabetically"""
        return self.name < other.name
    def __str__(self):
        """Returns a string representation of Tamagotchi"""
        return (self.name + ' - ' +
                'Age: ' + str(self.get_age().days) + ' days ' +
                'Food: ' + str(self.food))
    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
```

Polymorphism in Action

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

```
mimitchi = Tamagotchi("Mimitchi", date(2023, 8, 8))
```

```
sebiretchi = Tamagotchi("Sebiretchi", date(2023, 11, 1))
```

```
mimitchi < sebiretchi
```

```
True
```

```
sorted([mimitchi, sebiretchi])
```

```
[<__main__.Tamagotchi object at 0x78d8cad8b350>, <__main__.Tamagotchi object at 0x78d8b4fc5430>]
```

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

```
['Mimitchi - Age: 468 days Food: 0', 'Sebiretchi - Age: 383 days Food: 0']
```

Inheritance

Inheritance

- Classes allow 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

```python
class StatisticalTest:
    """Base class for Statistical tests"""
    def __init__(self, x, y = None, test_name = None):
        """
        Initialize the StatisticalTest.

        Parameters:
          - x: The first sample for the test.
          - y: The second sample for tests that compare two samples (default is None).
          - test_name: The name of the test as string (default is None).
        """
        self.x = x
        self.y = y
        self.test_name = test_name
        self.test_statistic = None
        self.p_value = None
    def __str__(self):
        """Return a string representation of the statistical test."""
        return f'{self.test_name}\n' \
               f'Test statistic: {self.test_statistic}\n' \
               f'P-value: {self.p_value}'
    def test(self):
        """
        Conduct the statistical test.

        This method performs the necessary calculations
        to obtain the test statistic and p-value based
        on the provided samples.

        Important: This method must be implemented in subclasses.
        """
        raise NotImplementedError
```

Superclass: Statistical Test

```
1 test = StatisticalTest([1, 2, 3])
```

```
1 type(test)
```

```
<class '__main__.StatisticalTest'>
```

```
1 print(test)
```

```
None
Test statistic: None
P-value: None
```

```
1 help(test.test)
```

```
Help on method test in module __main__:

test() method of __main__.StatisticalTest instance
    Conduct the statistical test.

    This method performs the necessary calculations
    to obtain the test statistic and p-value based
    on the provided samples.

    Important: This method must be implemented in subclasses.
```

```
1 test.test()
```

```
NotImplementedError
```

Subclass: T-test

When defining the subclass `TTest` we **override** the `__init__()` method.

```python
import numpy as np
from scipy.stats import t

class TTest(StatisticalTest):
    def __init__(self, x, y = None, test_name = None):
        super().__init__(x, y, test_name)

    def _prepare(self, x):
        """
        Prepare the sample for the t-test.
        """
        # Ensure x is NumPy array
        x = np.array(x)
        mean_x = np.mean(x)
        # Use ddof = 1 for sample variance in Python
        var_x = np.var(x, ddof = 1)
        n_x = len(x)
        se_x = np.sqrt(var_x / n_x)
        return x, mean_x, var_x, n_x, se_x
```

Subclass: 2-sample T-test

```python
class TTest_2samp(TTest):
    def __init__(self, x, y):
        super().__init__(x, y, 'Welch Two Sample t-test')

    def test(self):
        """
        Conduct two-sample t-test on the provided samples.
        """
        if self.y is None:
            raise ValueError("Two samples are required for an independent t-test.")
        x, mean_x, var_x, n_x, se_x = self._prepare(self.x)
        y, mean_y, var_y, n_y, se_y = self._prepare(self.y)
        se = np.sqrt(se_x ** 2 + se_y ** 2)
        # Calculate the t-statistic
        t_stat = (mean_x - mean_y) / se
        # Calculate the degrees of freedom
        df = se ** 4 / (se_x ** 4 / (n_x - 1) + se_y ** 4 / (n_y - 1))
        # Calculate the p-value
        p_val = 2 * (1 - t.cdf(abs(t_stat), df))
        # Set the test statistic and p-value attributes
        self.test_statistic = t_stat
        self.p_value = p_val
```

Subclass: Example

```
# Create a random number generator (RNG) object
rng  = np.random.default_rng(seed = 1000)
t_test_2samp = TTest_2samp(rng.normal(0, 1, 100), rng.normal(0, 1, 100))
```

```
help(t_test_2samp.test)
```

```
Help on method test in module __main__:

test() method of __main__.TTest_2samp instance
    Conduct two-sample t-test on the provided samples.
```

```
t_test_2samp.test()
```

```
print(t_test_2samp)
```

```
Welch Two Sample t-test
Test statistic: 0.5950642058053337
P-value: 0.5524817504963309
```

Inheritance Hierarchy

```
class TTest_1samp(TTest):
    pass
```

```
class ChisqTest(StatisticalTest):
    pass
```

```
t_test_1samp = TTest_1samp([1, 2, 3])
```

```
chisq_test = ChisqTest([1, 2, 3])
```

```
isinstance(t_test_1samp, StatisticalTest)
```

```
True
```

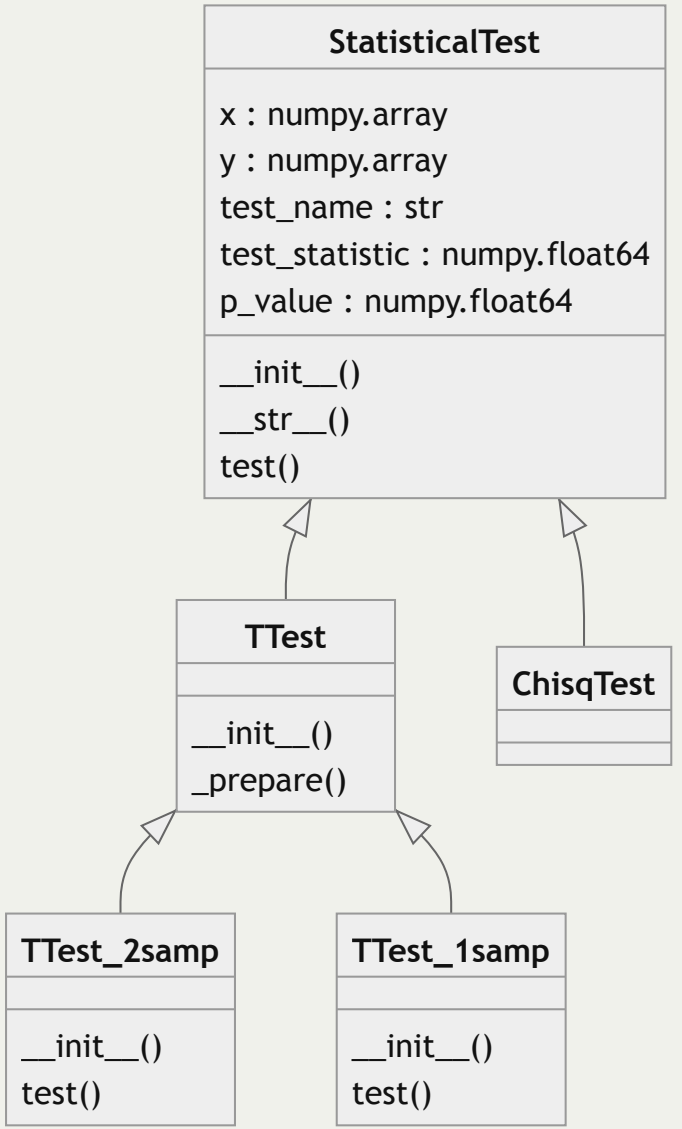
```
isinstance(chisq_test, StatisticalTest)
```

```
True
```

```
issubclass(TTest_2samp, TTest)
```

```
True
```

Inheritance Hierarchy: Diagram



OOP in Perspective

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.

OOP in R

- Somewhat confusingly, R has multiple (3+) OO systems.
- The closest to Python (**encapsulated** OOP) is offered by **RC** (reference classes).
- However, it is also not very widely used.
- More common approaches (**functional** OOP) are offered by **S3** and **S4** systems.
- These are based on **generic functions**.
- Functions that behave differently depending on the class of their arguments.
- S3 is simpler and more flexible, S4 is more structured and formal.

Next

- Tutorial: Python objects, classes and methods
- Assignment 4: Due at 12:00 on Monday, 25th November (submission on Blackboard)
- Next week: Complexity and performance