Introduction to Python III

Overview

- OOP: classes and objects
- Iterators and generators
- Context managers
- Decorators
- Development process
- The Python Package Index
- IPython
- Python scientific packages (just a mention):
  - `numpy`
  - `matplotlib`
  - `scipy`
  - `astropy`, `astroquery`

Introduction to Python III

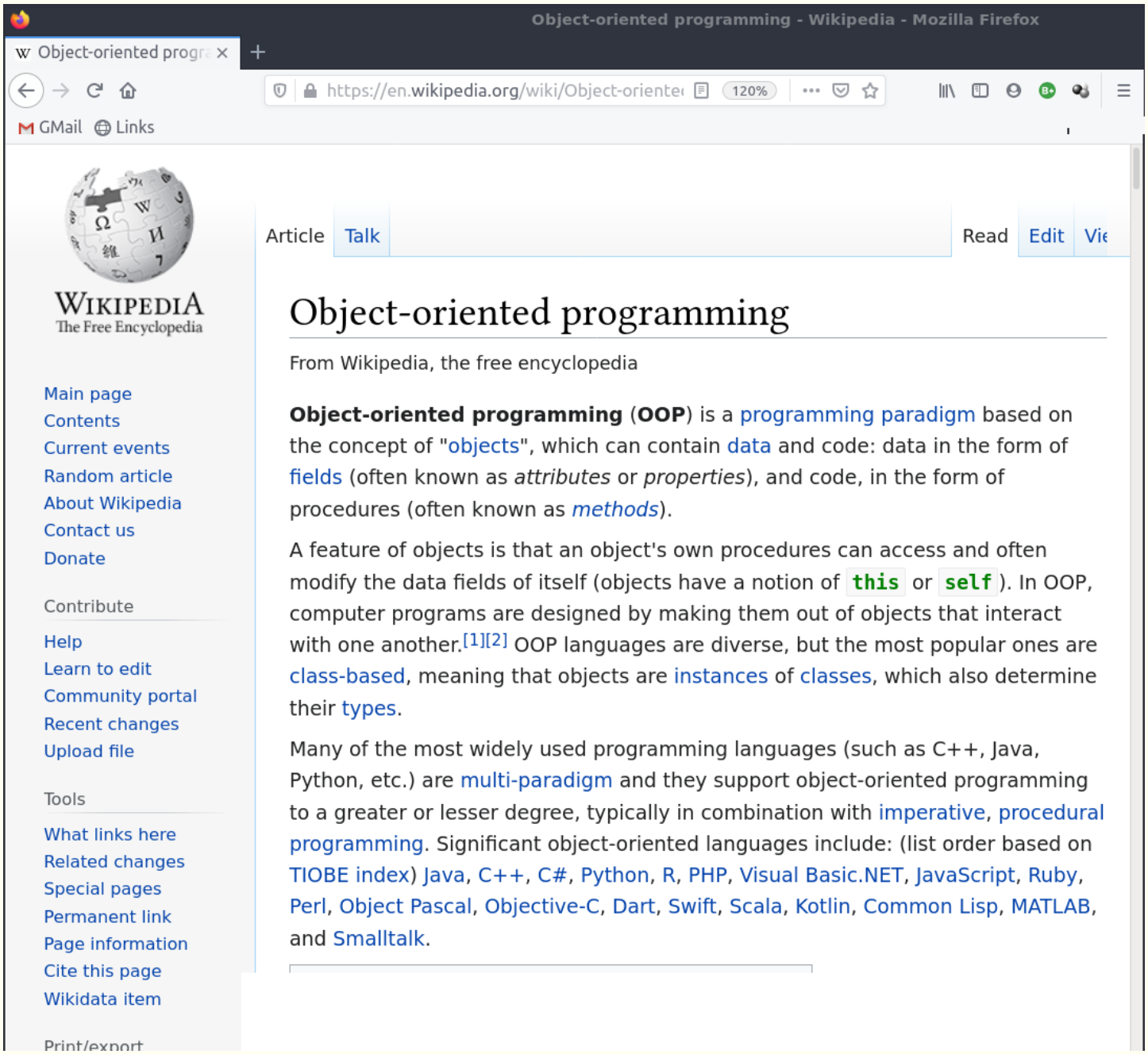
Overview

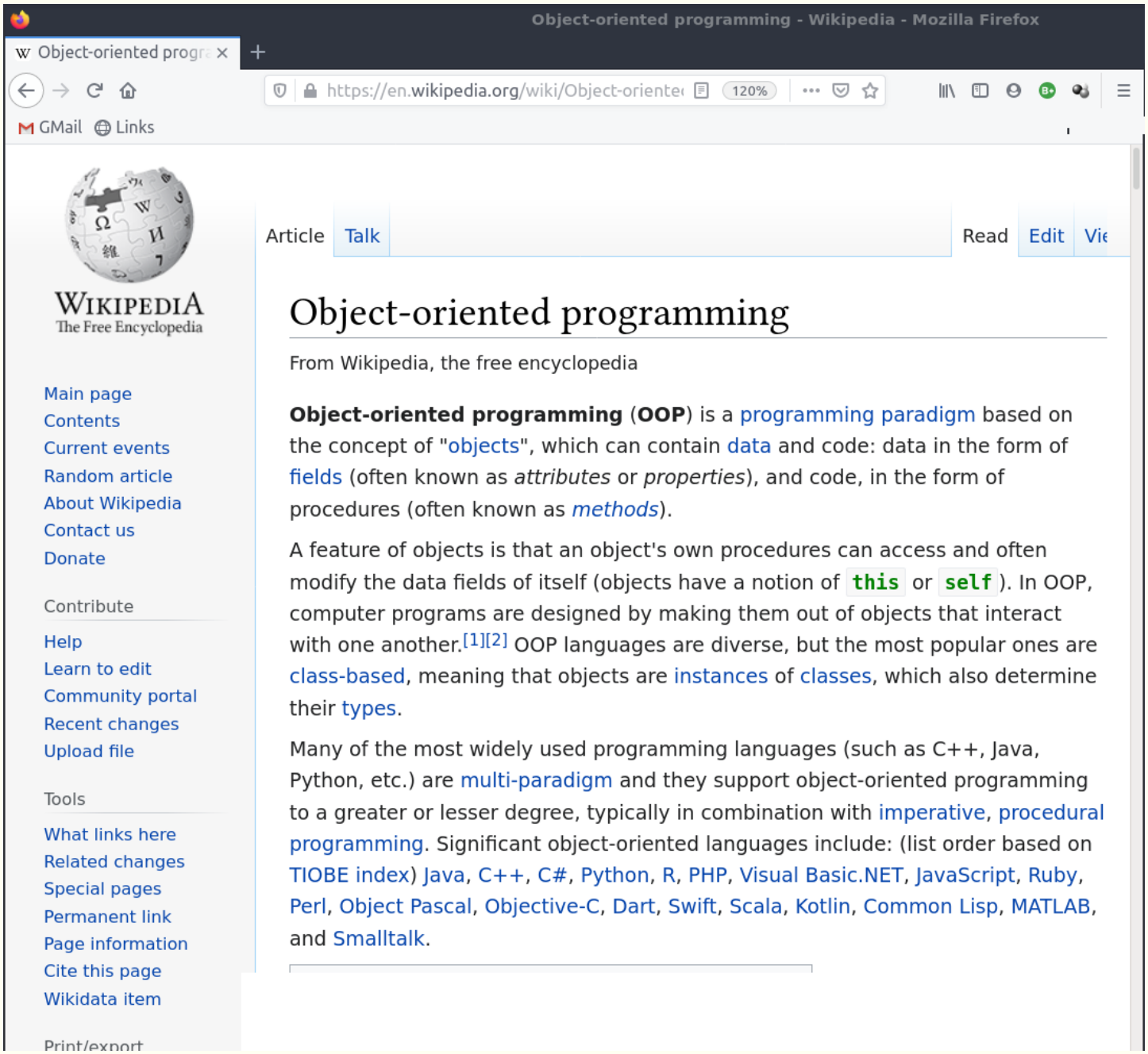
- OOP: classes and objects
- Iterators and generators
- Context managers
- Decorators
- Development process
- The Python Package Index
- IPython
- Python scientific packages (just a mention):
  - `numpy`
  - `matplotlib`
  - `scipy`
  - `astropy`, `astroquery`

→

Object Oriented Programming -1

Introduction III - 2




Object-oriented programming - Wikipedia - Mozilla Firefox
Object-oriented progra
https://en.wikipedia.org/wiki/Object-oriente
120%
GMail
Links
WIKIPEDIA
The Free Encyclopedia
Main page
Contents
Current events
Random article
About Wikipedia
Contact us
Donate
Contribute
Help
Learn to edit
Community portal
Recent changes
Upload file
Tools
What links here
Related changes
Special pages
Permanent link
Page information
Cite this page
Wikidata item
Print/export
Article
Talk
Read
Edit
Object-oriented programming
From Wikipedia, the free encyclopedia
Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which can contain data and code: data in the form of fields (often known as attributes or properties), and code, in the form of procedures (often known as methods).
A feature of objects is that an object's own procedures can access and often modify the data fields of itself (objects have a notion of this or self). In OOP, computer programs are designed by making them out of objects that interact with one another.[1][2] OOP languages are diverse, but the most popular ones are class-based, meaning that objects are instances of classes, which also determine their types.
Many of the most widely used programming languages (such as C++, Java, Python, etc.) are multi-paradigm and they support object-oriented programming to a greater or lesser degree, typically in combination with imperative, procedural programming. Significant object-oriented languages include: (list order based on TIOBE index) Java, C++, C#, Python, R, PHP, Visual Basic.NET, JavaScript, Ruby, Perl, Object Pascal, Objective-C, Dart, Swift, Scala, Kotlin, Common Lisp, MATLAB, and Smalltalk.

→

Classes and Objects -1

The **class** is the tool which allows a programmer to define her/his own objects.

Classes and Objects -1

The **class** is the tool which allows a programmer to define her/his own objects.

file: number.py

```
class Number:
    "An example of class: Number()"
    names=('zero','one','two','three','four',
           'five', 'six','seven','eight','nine')

    def __init__(self, n):
       self.name=Number.names[n]
       self.value=n

    def lower(self):
        return self.name

    def upper(self):
        return self.name.upper()
```

Classes and Objects -1

The **class** is the tool which allows a programmer to define her/his own objects.

file: number.py

```
class Number:                                    ← 1
   "An example of class: Number()"               ← 2
   names=('zero','one','two','three','four',     ← 3
          'five', 'six','seven','eight','nine')

   def __init__(self, n):                        ← 4,8
      self.name=Number.names[n]                  ← 5
      self.value=n                               ← 5

   def lower(self):                              ← 6,8
       return self.name

   def upper(self):                              ← 7,8
       return self.name.upper()
```

Classes and Objects -1

The **class** is the tool which allows a programmer to define her/his own objects.

file: number.py

```
class Number:                                    ← 1
   "An example of class: Number()"               ← 2
   names=('zero','one','two','three','four',     ← 3
          'five', 'six','seven','eight','nine')

   def __init__(self, n):                        ← 4,8
      self.name=Number.names[n]                  ← 5
      self.value=n                               ← 5

   def lower(self):                              ← 6,8
       return self.name

   def upper(self):                              ← 7,8
       return self.name.upper()
```

1. Definition of class **Number**

Classes and Objects -1

The **class** is the tool which allows a programmer to define her/his own objects.

file: `number.py`

```
class Number:                                    ← 1
   "An example of class: Number()"               ← 2
   names=('zero','one','two','three','four',     ← 3
          'five', 'six','seven','eight','nine')

   def __init__(self, n):                        ← 4,8
      self.name=Number.names[n]                  ← 5
      self.value=n                               ← 5

   def lower(self):                              ← 6,8
       return self.name

   def upper(self):                              ← 7,8
       return self.name.upper()
```

1. Definition of class **Number**
2. Class documentation string

Classes and Objects -1

The **class** is the tool which allows a programmer to define her/his own objects.

file: number.py

```
class Number:                                    ← 1
   "An example of class: Number()"               ← 2
   names=('zero','one','two','three','four',     ← 3
          'five', 'six','seven','eight','nine')

   def __init__(self, n):                        ← 4,8
      self.name=Number.names[n]                  ← 5
      self.value=n                               ← 5

   def lower(self):                              ← 6,8
       return self.name

   def upper(self):                              ← 7,8
       return self.name.upper()
```

1. Definition of class **Number**
2. Class documentation string
3. **names**: class attribute

Classes and Objects -1

The **class** is the tool which allows a programmer to define her/his own objects.

file: `number.py`

```
class Number:                                    ← 1
   "An example of class: Number()"               ← 2
   names=('zero','one','two','three','four',     ← 3
          'five', 'six','seven','eight','nine')

   def __init__(self, n):                        ← 4,8
      self.name=Number.names[n]                  ← 5
      self.value=n                               ← 5

   def lower(self):                              ← 6,8
       return self.name

   def upper(self):                              ← 7,8
       return self.name.upper()
```

1. Definition of class **Number**
2. Class documentation string
3. **names**: class attribute
4. Special method `__init__()`: **constructor**

Classes and Objects -1

The **class** is the tool which allows a programmer to define her/his own objects.

file: `number.py`

```
class Number:                                          ← 1
    "An example of class: Number()"                    ← 2
    names=('zero','one','two','three','four',          ← 3
           'five', 'six','seven','eight','nine')

    def __init__(self, n):                             ← 4,8
        self.name=Number.names[n]                      ← 5
        self.value=n                                   ← 5

    def lower(self):                                   ← 6,8
        return self.name

    def upper(self):                                   ← 7,8
        return self.name.upper()
```

1. Definition of class **Number**
2. Class documentation string
3. **names**: class attribute
4. Special method `__init__()`: **constructor**
5. `self.name`, `self.value`: instance attributes

Classes and Objects -1

The **class** is the tool which allows a programmer to define her/his own objects.

file: `number.py`

```
class Number:                                   ← 1
   "An example of class: Number()"              ← 2
   names=('zero','one','two','three','four',    ← 3
          'five', 'six','seven','eight','nine')

   def __init__(self, n):                       ← 4,8
      self.name=Number.names[n]                 ← 5
      self.value=n                              ← 5

   def lower(self):                             ← 6,8
       return self.name

   def upper(self):                             ← 7,8
       return self.name.upper()
```

1. Definition of class **Number**
2. Class documentation string
3. **names**: class attribute
4. Special method `__init__()`: **constructor**
5. `self.name`, `self.value`: instance attributes
6. Method: `lower()`

Classes and Objects -1

The **class** is the tool which allows a programmer to define her/his own objects.

file: `number.py`

```
class Number:                                    ← 1
   "An example of class: Number()"               ← 2
   names=('zero','one','two','three','four',     ← 3
          'five', 'six','seven','eight','nine')

   def __init__(self, n):                        ← 4,8
      self.name=Number.names[n]                  ← 5
      self.value=n                               ← 5

   def lower(self):                              ← 6,8
       return self.name

   def upper(self):                              ← 7,8
       return self.name.upper()
```

1. Definition of class **Number**
2. Class documentation string
3. **names**: class attribute
4. Special method `__init__()`: **constructor**
5. `self.name`, `self.value`: instance attributes
6. Method: `lower()`
7. Method: `upper()`

Classes and Objects -1

The **class** is the tool which allows a programmer to define her/his own objects.

file: `number.py`

```
class Number:                                          ← 1
   "An example of class: Number()"                     ← 2
   names=('zero','one','two','three','four',           ← 3
          'five', 'six','seven','eight','nine')

   def __init__(self, n):                              ← 4,8
      self.name=Number.names[n]                        ← 5
      self.value=n                                     ← 5

   def lower(self):                                    ← 6,8
       return self.name

   def upper(self):                                    ← 7,8
       return self.name.upper()
```

1. Definition of class **Number**
2. Class documentation string
3. **names**: class attribute
4. Special method `__init__()`: **constructor**
5. `self.name`, `self.value`: instance attributes
6. Method: `lower()`
7. Method: `upper()`
8. **Note**: in all methods the first argument (strictly required) refers to the instance (it's usually named: `self`)

Classes and Objects -1

The **class** is the tool which allows a programmer to define her/his own objects.

file: `number.py`

```
class Number:                                          ← 1
   "An example of class: Number()"                     ← 2
   names=('zero','one','two','three','four',           ← 3
          'five', 'six','seven','eight','nine')

   def __init__(self, n):                              ← 4,8
      self.name=Number.names[n]                        ← 5
      self.value=n                                     ← 5

   def lower(self):                                    ← 6,8
       return self.name

   def upper(self):                                    ← 7,8
       return self.name.upper()
```

1. Definition of class **Number**
2. Class documentation string
3. **names**: class attribute
4. Special method `__init__()`: **constructor**
5. `self.name`, `self.value`: instance attributes
6. Method: `lower()`
7. Method: `upper()`
8. **Note**: in all methods the first argument (strictly required) refers to the instance (it's usually named: `self`)

→

Classes and Objects -2

How to use class `Number`:

```
>>> from number import Number
>>> a=Number(2)
>>> b=Number(3)
>>> a
<number.Number object at 0x7f9153a695f8>
>>> b
<number.Number object at 0x7f9153a695c0>
>>> a.value
2
>>> b.upper()
'THREE'
>>> a.lower()
'two'
>>>
```

Classes and Objects -2

How to use class `Number`:

```
>>> from number import Number
>>> a=Number(2)
>>> b=Number(3)
>>> a
<number.Number object at 0x7f9153a695f8>
>>> b
<number.Number object at 0x7f9153a695c0>
>>> a.value
2
>>> b.upper()
'THREE'
>>> a.lower()
'two'
>>>
```

1,5

2

3,5

4,5

Classes and Objects -2

How to use class `Number`:

```
>>> from number import Number
>>> a=Number(2)
>>> b=Number(3)
>>> a
<number.Number object at 0x7f9153a695f8>
>>> b
<number.Number object at 0x7f9153a695c0>
>>> a.value
2
>>> b.upper()
'THREE'
>>> a.lower()
'two'
>>>
```

1,5

2

3,5

4,5

1. Creating two instances (objects) of class `Number` (equivalent to calling the `__init__()` method)

Classes and Objects -2

How to use class `Number`:

```
>>> from number import Number
>>> a=Number(2)
>>> b=Number(3)
>>> a
<number.Number object at 0x7f9153a695f8>
>>> b
<number.Number object at 0x7f9153a695c0>
>>> a.value
2
>>> b.upper()
'THREE'
>>> a.lower()
'two'
>>>
```

1,5

2

3,5

4,5

1. Creating two instances (objects) of class `Number` (equivalent to calling the `__init__()` method)
2. Using object's attribute `value`

Classes and Objects -2

How to use class `Number`:

```
>>> from number import Number
>>> a=Number(2)
>>> b=Number(3)
>>> a
<number.Number object at 0x7f9153a695f8>
>>> b
<number.Number object at 0x7f9153a695c0>
>>> a.value
2
>>> b.upper()
'THREE'
>>> a.lower()
'two'
>>>
```

1,5

2

3,5

4,5

1. Creating two instances (objects) of class `Number` (equivalent to calling the `__init__()` method)
2. Using object's attribute `value`
3. Calling object's method `upper()`

Classes and Objects -2

How to use class `Number`:

```
>>> from number import Number
>>> a=Number(2)
>>> b=Number(3)
>>> a
<number.Number object at 0x7f9153a695f8>
>>> b
<number.Number object at 0x7f9153a695c0>
>>> a.value
2
>>> b.upper()
'THREE'
>>> a.lower()
'two'
>>>
```

1,5

2

3,5

4,5

1. Creating two instances (objects) of class `Number` (equivalent to calling the `__init__()` method)
2. Using object's attribute `value`
3. Calling object's method `upper()`
4. Calling object's method `lower()`

Classes and Objects -2

How to use class `Number`:

```
>>> from number import Number
>>> a=Number(2)
>>> b=Number(3)
>>> a
<number.Number object at 0x7f9153a695f8>
>>> b
<number.Number object at 0x7f9153a695c0>
>>> a.value
2
>>> b.upper()
'THREE'
>>> a.lower()
'two'
>>>
```

1,5

2

3,5

4,5

1. Creating two instances (objects) of class `Number` (equivalent to calling the `__init__()` method)
2. Using object's attribute `value`
3. Calling object's method `upper()`
4. Calling object's method `lower()`
5. **Note**: when calling a method the first required argument is implicit

Classes and Objects -2

How to use class `Number`:

```
>>> from number import Number
>>> a=Number(2)
>>> b=Number(3)
>>> a
<number.Number object at 0x7f9153a695f8>
>>> b
<number.Number object at 0x7f9153a695c0>
>>> a.value
2
>>> b.upper()
'THREE'
>>> a.lower()
'two'
>>>
```

1,5

2

3,5

4,5

1. Creating two instances (objects) of class `Number` (equivalent to calling the `__init__()` method)
2. Using object's attribute `value`
3. Calling object's method `upper()`
4. Calling object's method `lower()`
5. **Note**: when calling a method the first required argument is implicit

→

Derived classes -1

Proceeding further with the previous example:

Derived classes -1

Proceeding further with the previous example:

file: `numberplus.py`

```
from number import Number

class NumberPlus(Number):
    "Number with addition"
    def plus(self, x):
        return NumberPlus(self.value+x.value)

    def upper(self):
        return self.name.capitalize()
```

Derived classes -1

Proceeding further with the previous example:

file: `numberplus.py`

```
from number import Number

class NumberPlus(Number):          <-- 1, 2
    "Number with addition"
    def plus(self, x):             <-- 3
        return NumberPlus(self.value+x.value)

    def upper(self):               <-- 4
        return self.name.capitalize()
```

Derived classes -1

Proceeding further with the previous example:

file: `numberplus.py`

```
from number import Number

class NumberPlus(Number):          <-- 1, 2
    "Number with addition"
    def plus(self, x):             <-- 3
        return NumberPlus(self.value+x.value)

    def upper(self):               <-- 4
        return self.name.capitalize()
```

1. Here we define a new class **derived** from `Number`

Derived classes -1

Proceeding further with the previous example:

file: `numberplus.py`

```
from number import Number

class NumberPlus(Number):          <-- 1, 2
    "Number with addition"
    def plus(self, x):             <-- 3
        return NumberPlus(self.value+x.value)

    def upper(self):               <-- 4
        return self.name.capitalize()
```

1. Here we define a new class **derived** from `Number`
2. `NumberPlus` is a sub-class of `Number`

Derived classes -1

Proceeding further with the previous example:

file: `numberplus.py`

```
from number import Number

class NumberPlus(Number):          <-- 1, 2
    "Number with addition"
    def plus(self, x):             <-- 3
        return NumberPlus(self.value+x.value)

    def upper(self):               <-- 4
        return self.name.capitalize()
```

1. Here we define a new class **derived** from `Number`
2. `NumberPlus` is a sub-class of `Number`
3. Implementation of a new method: `plus`

Derived classes -1

Proceeding further with the previous example:

file: `numberplus.py`

```
from number import Number

class NumberPlus(Number):          <-- 1, 2
    "Number with addition"
    def plus(self, x):             <-- 3
        return NumberPlus(self.value+x.value)

    def upper(self):               <-- 4
        return self.name.capitalize()
```

1. Here we define a new class **derived** from `Number`
2. `NumberPlus` is a sub-class of `Number`
3. Implementation of a new method: `plus`
4. Re-implementation (*overload*) of method: `upper()`

Derived classes -1

Proceeding further with the previous example:

file: `numberplus.py`

```
from number import Number

class NumberPlus(Number):          <-- 1, 2
    "Number with addition"
    def plus(self, x):             <-- 3
        return NumberPlus(self.value+x.value)

    def upper(self):               <-- 4
        return self.name.capitalize()
```

1. Here we define a new class **derived** from `Number`
2. `NumberPlus` is a sub-class of `Number`
3. Implementation of a new method: `plus`
4. Re-implementation (*overload*) of method: `upper()`

→

Derived classes -2

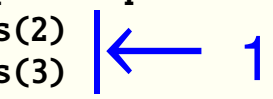
Let's use the new class:

```
>>> from numberplus import NumberPlus
>>> a=NumberPlus(2)
>>> b=NumberPlus(3)
>>> print(a)
<numberext.NumberPlus object at 0x7fe17970dc88>
>>> print(b)
<numberext.NumberPlus object at 0x7fe17970dc50>
>>> c=a.plus(b)
>>> print(c)
<numberplus.NumberPlus object at 0x7ffa15698710>
>>> c.value
5
>>> c.upper()
'Five'
>>>
```

Derived classes -2

Let's use the new class:

```
>>> from numberplus import NumberPlus
>>> a=NumberPlus(2)
>>> b=NumberPlus(3)
>>> print(a)
<numberext.NumberPlus object at 0x7fe17970dc88>
>>> print(b)
<numberext.NumberPlus object at 0x7fe17970dc50>
>>> c=a.plus(b)
>>> print(c)
<numberplus.NumberPlus object at 0x7ffa15698710>
>>> c.value
5
>>> c.upper()
'Five'
>>>
```



Derived classes -2

Let's use the new class:

```
>>> from numberplus import NumberPlus
>>> a=NumberPlus(2)
>>> b=NumberPlus(3)
>>> print(a)
<numberext.NumberPlus object at 0x7fe17970dc88>
>>> print(b)
<numberext.NumberPlus object at 0x7fe17970dc50>
>>> c=a.plus(b)
>>> print(c)
<numberplus.NumberPlus object at 0x7ffa15698710>
>>> c.value
5
>>> c.upper()
'Five'
>>>
```

1 ← 1

2 ← 2

3 ← 3

4 ← 4

1. Creating two instances of class `NumberPlus`

Derived classes -2

Let's use the new class:

```
>>> from numberplus import NumberPlus
>>> a=NumberPlus(2)
>>> b=NumberPlus(3)
>>> print(a)
<numberext.NumberPlus object at 0x7fe17970dc88>
>>> print(b)
<numberext.NumberPlus object at 0x7fe17970dc50>
>>> c=a.plus(b)
>>> print(c)
<numberplus.NumberPlus object at 0x7ffa15698710>
>>> c.value
5
>>> c.upper()
'Five'
>>>
```

1 ← 1

2 ← 2

3 ← 3

4 ← 4

1. Creating two instances of class `NumberPlus`
2. The `plus()` method creates a new instance of `NumberPlus`

Derived classes -2

Let's use the new class:

```
>>> from numberplus import NumberPlus
>>> a=NumberPlus(2)
>>> b=NumberPlus(3)
>>> print(a)
<numberext.NumberPlus object at 0x7fe17970dc88>
>>> print(b)
<numberext.NumberPlus object at 0x7fe17970dc50>
>>> c=a.plus(b)
>>> print(c)
<numberplus.NumberPlus object at 0x7ffa15698710>
>>> c.value
5
>>> c.upper()
'Five'
>>>
```

1 ← 1

2 ← 2

3 ← 3

4 ← 4

1. Creating two instances of class `NumberPlus`
2. The `plus()` method creates a new instance of `NumberPlus`
3. Its value is the sum of the values of the two objects.

Derived classes -2

Let's use the new class:

```
>>> from numberplus import NumberPlus
>>> a=NumberPlus(2)
>>> b=NumberPlus(3)        <-- 1
>>> print(a)
<numberext.NumberPlus object at 0x7fe17970dc88>
>>> print(b)
<numberext.NumberPlus object at 0x7fe17970dc50>
>>> c=a.plus(b)            <-- 2
>>> print(c)
<numberplus.NumberPlus object at 0x7ffa15698710>
>>> c.value                <-- 3
5
>>> c.upper()              <-- 4
'Five'
>>>
```

1. Creating two instances of class `NumberPlus`
2. The `plus()` method creates a new instance of `NumberPlus`
3. Its value is the sum of the values of the two objects.
4. The `upper()` method has been redefined.

Derived classes -2

Let's use the new class:

```
>>> from numberplus import NumberPlus
>>> a=NumberPlus(2)
>>> b=NumberPlus(3)        <-- 1
>>> print(a)
<numberext.NumberPlus object at 0x7fe17970dc88>
>>> print(b)
<numberext.NumberPlus object at 0x7fe17970dc50>
>>> c=a.plus(b)            <-- 2
>>> print(c)
<numberplus.NumberPlus object at 0x7ffa15698710>
>>> c.value                <-- 3
5
>>> c.upper()              <-- 4
'Five'
>>>
```

1. Creating two instances of class `NumberPlus`
2. The `plus()` method creates a new instance of `NumberPlus`
3. Its value is the sum of the values of the two objects.
4. The `upper()` method has been redefined.

N.B.: The class `Number` and its derivatives implement very little of integer arithmetic!

Derived classes -2

Let's use the new class:

```
>>> from numberplus import NumberPlus
>>> a=NumberPlus(2)
>>> b=NumberPlus(3)
>>> print(a)
<numberext.NumberPlus object at 0x7fe17970dc88>
>>> print(b)
<numberext.NumberPlus object at 0x7fe17970dc50>
>>> c=a.plus(b)
>>> print(c)
<numberplus.NumberPlus object at 0x7ffa15698710>
>>> c.value
5
>>> c.upper()
'Five'
>>>
```

1 ← 1
2 ← 2
3 ← 3
4 ← 4

1. Creating two instances of class `NumberPlus`
2. The `plus()` method creates a new instance of `NumberPlus`
3. Its value is the sum of the values of the two objects.
4. The `upper()` method has been redefined.

N.B.: The class `Number` and its derivatives implement very little of integer arithmetic!

→

Emulating standard types -1

Python applies the **duck typing** principle:

Emulating standard types -1

Python applies the **duck typing** principle:

> If it walks like a duck and it quacks like a duck, then it must be a duck

E.g.: an object is (compatible with) an integer if provides all the methods which are specific of integers.

Emulating standard types -1

Python applies the **duck typing** principle:

> If it walks like a duck and it quacks like a duck, then it must be a duck

E.g.: an object is (compatible with) an integer if provides all the methods which are specific of integers.

```
>>> from numberplus import NumberPlus
>>> a=NumberPlus(2)
>>> b=NumberPlus(3)
>>> a+b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NumberPlus' and 'NumberPlus'
>>>
```

Emulating standard types -1

Python applies the **duck typing** principle:

> If it walks like a duck and it quacks like a duck, then it must be a duck

E.g.: an object is (compatible with) an integer if provides all the methods which are specific of integers.

```
>>> from numberplus import NumberPlus
>>> a=NumberPlus(2)
>>> b=NumberPlus(3)
>>> a+b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NumberPlus' and 'NumberPlus'
>>>
```

Our NumberPlus class doesn't add like an integer

Emulating standard types -1

Python applies the **duck typing** principle:

> If it walks like a duck and it quacks like a duck, then it must be a duck

E.g.: an object is (compatible with) an integer if provides all the methods which are specific of integers.

```
>>> from numberplus import NumberPlus
>>> a=NumberPlus(2)
>>> b=NumberPlus(3)
>>> a+b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NumberPlus' and 'NumberPlus'
>>>
```

Our NumberPlus class doesn't add like an integer

Let's look into an object of type integer:

Emulating standard types -1

Python applies the **duck typing** principle:

> If it walks like a duck and it quacks like a duck, then it must be a duck

E.g.: an object is (compatible with) an integer if provides all the methods which are specific of integers.

```
>>> from numberplus import NumberPlus
>>> a=NumberPlus(2)
>>> b=NumberPlus(3)
>>> a+b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NumberPlus' and 'NumberPlus'
>>>
```

Our NumberPlus class doesn't add like an integer

Let's look into an object of type integer:

```
>>> a = 1
>>> dir(a)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__',
'__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__',
   ....
```

Emulating standard types -1

Python applies the **duck typing** principle:

> If it walks like a duck and it quacks like a duck, then it must be a duck

E.g.: an object is (compatible with) an integer if provides all the methods which are specific of integers.

```
>>> from numberplus import NumberPlus
>>> a=NumberPlus(2)
>>> b=NumberPlus(3)
>>> a+b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NumberPlus' and 'NumberPlus'
>>>
```

Our NumberPlus class doesn't add like an integer

Let's look into an object of type integer:

```
>>> a = 1
>>> dir(a)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__',
'__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__',
   ....
```

Emulating standard types -1

Python applies the **duck typing** principle:

> If it walks like a duck and it quacks like a duck, then it must be a duck

E.g.: an object is (compatible with) an integer if provides all the methods which are specific of integers.

```
>>> from numberplus import NumberPlus
>>> a=NumberPlus(2)
>>> b=NumberPlus(3)
>>> a+b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NumberPlus' and 'NumberPlus'
>>>
```

Our NumberPlus class doesn't add like an integer

Let's look into an object of type integer:

```
>>> a = 1
>>> dir(a)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__',
'__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__',
   ....
```

→

Emulating standard types -2

Let's see how we can improve our class to support addition using the + sign.

file: `numberplus1.py` (class `NumberPlus` revised)

```
from number import Number

class NumberPlus(Number):
    "Number with addition"
    def __add__(self, x):
        return NumberPlus(self.value+x.value)

    def __str__(self):
        return self.name

    def upper(self):
        return self.name.capitalize()
```

Emulating standard types -2

Let's see how we can improve our class to support addition using the **+** sign.

file: `numberplus1.py` (class `NumberPlus` revised)

```
from number import Number

class NumberPlus(Number):
    "Number with addition"
    def __add__(self, x):          <-- 1,3
        return NumberPlus(self.value+x.value)

    def __str__(self):             <-- 2,4
        return self.name

    def upper(self):
        return self.name.capitalize()
```

Emulating standard types -2

Let's see how we can improve our class to support addition using the + sign.

file: `numberplus1.py` (class `NumberPlus` revised)

```
from number import Number

class NumberPlus(Number):
    "Number with addition"
    def __add__(self, x):          <-- 1,3
        return NumberPlus(self.value+x.value)

    def __str__(self):             <-- 2,4
        return self.name

    def upper(self):
        return self.name.capitalize()
```

1. The `plus()` method has been renamed: `__add__()`

Emulating standard types -2

Let's see how we can improve our class to support addition using the **+** sign.

file: `numberplus1.py` (class `NumberPlus` revised)

```
from number import Number

class NumberPlus(Number):
    "Number with addition"
    def __add__(self, x):          <-- 1,3
        return NumberPlus(self.value+x.value)

    def __str__(self):             <-- 2,4
        return self.name

    def upper(self):
        return self.name.capitalize()
```

1. The `plus()` method has been renamed: `__add__()`
2. A new method has been added: `__str__()`

Emulating standard types -2

Let's see how we can improve our class to support addition using the + sign.

file: `numberplus1.py` (class `NumberPlus` revised)

```
from number import Number

class NumberPlus(Number):
    "Number with addition"
    def __add__(self, x):          <-- 1,3
        return NumberPlus(self.value+x.value)

    def __str__(self):             <-- 2,4
        return self.name

    def upper(self):
        return self.name.capitalize()
```

1. The `plus()` method has been renamed: `__add__()`
2. A new method has been added: `__str__()`
3. `__add__()` is the method used implicitly in expressions like: `a+b`

Emulating standard types -2

Let's see how we can improve our class to support addition using the + sign.

file: `numberplus1.py` (class `NumberPlus` revised)

```
from number import Number

class NumberPlus(Number):
    "Number with addition"
    def __add__(self, x):          <-- 1,3
        return NumberPlus(self.value+x.value)

    def __str__(self):             <-- 2,4
        return self.name

    def upper(self):
        return self.name.capitalize()
```

1. The `plus()` method has been renamed: `__add__()`
2. A new method has been added: `__str__()`
3. `__add__()` is the method used implicitly in expressions like: `a+b`
4. `__str__()` is the method used implicitly in calls like: `print(a)`

Emulating standard types -2

Let's see how we can improve our class to support addition using the + sign.

file: `numberplus1.py` (class `NumberPlus` revised)

```
from number import Number

class NumberPlus(Number):
    "Number with addition"
    def __add__(self, x):          <-- 1,3
        return NumberPlus(self.value+x.value)

    def __str__(self):             <-- 2,4
        return self.name

    def upper(self):
        return self.name.capitalize()
```

1. The `plus()` method has been renamed: `__add__()`
2. A new method has been added: `__str__()`
3. `__add__()` is the method used implicitly in expressions like: `a+b`
4. `__str__()` is the method used implicitly in calls like: `print(a)`

→

Emulating standard types -3

And here's what we get:

```
>>> from numberplus1 import NumberPlus
>>> a=NumberPlus(2)
>>> b=NumberPlus(5)
>>> print(a)
two
>>> print(a+b)
seven
>>> a
<numberplus1.NumberPlus object at 0x7f6bbcdf3748>
>>>
```

Emulating standard types -3

And here's what we get:

```
>>> from numberplus1 import NumberPlus
>>> a=NumberPlus(2)
>>> b=NumberPlus(5)
>>> print(a) ← 1
two
>>> print(a+b) ← 2,3
seven
>>> a ← 4
<numberplus1.NumberPlus object at 0x7f6bbcdf3748>
>>>
```

1. The built-in function `print()` calls the standard method `__str__` to get the value of an object

Emulating standard types -3

And here's what we get:

```
>>> from numberplus1 import NumberPlus
>>> a=NumberPlus(2)
>>> b=NumberPlus(5)
>>> print(a) ⟵ 1
two
>>> print(a+b) ⟵ 2,3
seven
>>> a ⟵ 4
<numberplus1.NumberPlus object at 0x7f6bbcdf3748>
>>>
```

1. The built-in function `print()` calls the standard method `__str__` to get the value of an object
2. The + sign in the expression calls the `__add__()` method of the first object, i.e.:

   `a+b` is equivalent to: `a.__add__(b)`

Emulating standard types -3

And here's what we get:

```
>>> from numberplus1 import NumberPlus
>>> a=NumberPlus(2)
>>> b=NumberPlus(5)
>>> print(a) ← 1
two
>>> print(a+b) ← 2,3
seven
>>> a ← 4
<numberplus1.NumberPlus object at 0x7f6bbcdf3748>
>>>
```

1. The built-in function `print()` calls the standard method `__str__` to get the value of an object
2. The + sign in the expression calls the `__add__()` method of the first object, i.e.:

   `a+b` is equivalent to: `a.__add__(b)`
3. As above, `print()` calls `__str__()` on the value returned by `__add__()` (which is an object of type `NumberPlus`)

Emulating standard types -3

And here's what we get:

```
>>> from numberplus1 import NumberPlus
>>> a=NumberPlus(2)
>>> b=NumberPlus(5)
>>> print(a) ← 1
two
>>> print(a+b) ← 2,3
seven
>>> a ← 4
<numberplus1.NumberPlus object at 0x7f6bbcdf3748>
>>>
```

1. The built-in function `print()` calls the standard method `__str__` to get the value of an object
2. The + sign in the expression calls the `__add__()` method of the first object, i.e.:

   `a+b` is equivalent to: `a.__add__(b)`
3. As above, `print()` calls `__str__()` on the value returned by `__add__()` (which is an object of type `NumberPlus`)
4. The python interpreter in interactive mode, when showing the value of an object, calls the standard method `__repr__()`

Emulating standard types -3

And here's what we get:

```
>>> from numberplus1 import NumberPlus
>>> a=NumberPlus(2)
>>> b=NumberPlus(5)
>>> print(a) ← 1
two
>>> print(a+b) ← 2,3
seven
>>> a ← 4
<numberplus1.NumberPlus object at 0x7f6bbcdf3748>
>>>
```

1. The built-in function `print()` calls the standard method `__str__` to get the value of an object
2. The + sign in the expression calls the `__add__()` method of the first object, i.e.:

   `a+b` is equivalent to: `a.__add__(b)`
3. As above, `print()` calls `__str__()` on the value returned by `__add__()` (which is an object of type `NumberPlus`)
4. The python interpreter in interactive mode, when showing the value of an object, calls the standard method `__repr__()`

→

Iterables & iterators -1

Iterables & iterators -1

- Iterables are objects on which a `for` loop can iterate

Iterables & iterators -1

- Iterables are objects on which a `for` loop can iterate
- An objects "quacks" as an iterable if it provides an `__iter__` method providing an *iterator*

Iterables & iterators -1

- Iterables are objects on which a `for` loop can iterate
- An objects "quacks" as an iterable if it provides an `__iter__` method providing an *iterator*
- A simple example of iterable: the `tuple`

```
>>> a=(1,2,3,4,5)
>>> type(a)
<class 'tuple'>
>>> dir(a)
['__add__', ....., '__init__', ...., '__iter__', ..., 'count', 'index']
>>> for k in a:
...   print(k, end=" - ")
...
1 - 2 - 3 - 4 - 5 - >>>
>>>
```

Iterables & iterators -1

- Iterables are objects on which a `for` loop can iterate
- An objects "quacks" as an iterable if it provides an `__iter__` method providing an *iterator*
- A simple example of iterable: the `tuple`

```
>>> a=(1,2,3,4,5)
>>> type(a)
<class 'tuple'>
>>> dir(a)
['__add__', ....., '__init__', ...., '__iter__', ..., 'count', 'index']
>>> for k in a:
...   print(k, end=" - ")
...
1 - 2 - 3 - 4 - 5 - >>>
>>>
```

Iterables & iterators -1

- Iterables are objects on which a `for` loop can iterate
- An objects "quacks" as an iterable if it provides an `__iter__` method providing an *iterator*
- A simple example of iterable: the `tuple`

```
>>> a=(1,2,3,4,5)
>>> type(a)
<class 'tuple'>
>>> dir(a)
['__add__', ....., '__init__', ...., '__iter__', ..., 'count', 'index']
>>> for k in a:
...   print(k, end=" - ")
...
1 - 2 - 3 - 4 - 5 - >>>
>>>
```

Behind the scenes of the for loop:

Iterables & iterators -1

- Iterables are objects on which a `for` loop can iterate
- An objects "quacks" as an iterable if it provides an `__iter__` method providing an *iterator*
- A simple example of iterable: the `tuple`

```
>>> a=(1,2,3,4,5)
>>> type(a)
<class 'tuple'>
>>> dir(a)
['__add__', ....., '__init__', ...., '__iter__', ..., 'count', 'index']
>>> for k in a:
...   print(k, end=" - ")
...
1 - 2 - 3 - 4 - 5 - >>>
>>>
```

Behind the scenes of the for loop:

1. `a.__iter__()` is called to get an **iterator**

Iterables & iterators -1

- Iterables are objects on which a `for` loop can iterate
- An objects "quacks" as an iterable if it provides an `__iter__` method providing an *iterator*
- A simple example of iterable: the `tuple`

```
>>> a=(1,2,3,4,5)
>>> type(a)
<class 'tuple'>
>>> dir(a)
['__add__', ....., '__init__', ...., '__iter__', ..., 'count', 'index']
>>> for k in a:
...   print(k, end=" - ")
...
1 - 2 - 3 - 4 - 5 - >>>
>>>
```

Behind the scenes of the for loop:

1. `a.__iter__()` is called to get an **iterator**
2. `k=`*`iterator`*`.__next__()` is executed repeatedly until a `StopIteration` exception is raised

Iterables & iterators -1

- Iterables are objects on which a `for` loop can iterate
- An objects "quacks" as an iterable if it provides an `__iter__` method providing an *iterator*
- A simple example of iterable: the `tuple`

```
>>> a=(1,2,3,4,5)
>>> type(a)
<class 'tuple'>
>>> dir(a)
['__add__', ....., '__init__', ...., '__iter__', ..., 'count', 'index']
>>> for k in a:
...   print(k, end=" - ")
...
1 - 2 - 3 - 4 - 5 - >>>
>>>
```

1,2

Behind the scenes of the for loop:

1. `a.__iter__()` is called to get an **iterator**
2. `k=`*`iterator`*`.__next__()` is executed repeatedly until a `StopIteration` exception is raised

→

Iterables & iterators -2

Let's make a class wich can be used as an iterable

file: `fibo1.py`

```
class Fibo:
    "Iterator for the Fibonacci series"
    def __init__(self, maxv):
        self.maxv = maxv

    def __iter__(self):
        self.a = 0
        self.b = 1
        return self

    def __next__(self):
        fib = self.a
        if fib > self.maxv:
            raise StopIteration
        self.a, self.b = self.b, self.a + self.b
        return fib
```

Iterables & iterators -2

Let's make a class wich can be used as an iterable

file: `fibo1.py`

```
class Fibo:
    "Iterator for the Fibonacci series"
    def __init__(self, maxv):
        self.maxv = maxv

    def __iter__(self):          <-- 1
        self.a = 0
        self.b = 1
        return self              <-- 2

    def __next__(self):          <-- 3
        fib = self.a
        if fib > self.maxv:
            raise StopIteration  <-- 4
        self.a, self.b = self.b, self.a + self.b
        return fib
```

Iterables & iterators -2

Let's make a class wich can be used as an iterable

file: `fibo1.py`

```
class Fibo:
    "Iterator for the Fibonacci series"
    def __init__(self, maxv):
        self.maxv = maxv

    def __iter__(self):          <-- 1
        self.a = 0
        self.b = 1
        return self              <-- 2

    def __next__(self):          <-- 3
        fib = self.a
        if fib > self.maxv:
            raise StopIteration  <-- 4
        self.a, self.b = self.b, self.a + self.b
        return fib
```

1. This class is provided with an `__iter__()` method

Iterables & iterators -2

Let's make a class wich can be used as an iterable

file: `fibo1.py`

```
class Fibo:
    "Iterator for the Fibonacci series"
    def __init__(self, maxv):
        self.maxv = maxv

    def __iter__(self):          <-- 1
        self.a = 0
        self.b = 1
        return self              <-- 2

    def __next__(self):          <-- 3
        fib = self.a
        if fib > self.maxv:
            raise StopIteration  <-- 4
        self.a, self.b = self.b, self.a + self.b
        return fib
```

1. This class is provided with an `__iter__()` method
2. Which returns itself

Iterables & iterators -2

Let's make a class wich can be used as an iterable

file: `fibo1.py`

```
class Fibo:
    "Iterator for the Fibonacci series"
    def __init__(self, maxv):
        self.maxv = maxv

    def __iter__(self):          <-- 1
        self.a = 0
        self.b = 1
        return self              <-- 2

    def __next__(self):          <-- 3
        fib = self.a
        if fib > self.maxv:
            raise StopIteration  <-- 4
        self.a, self.b = self.b, self.a + self.b
        return fib
```

1. This class is provided with an `__iter__()` method
2. Which returns itself
3. The returned object is provided with a `__next__()` method and thus it is an **iterator**

Iterables & iterators -2

Let's make a class wich can be used as an iterable

file: `fibo1.py`

```
class Fibo:
    "Iterator for the Fibonacci series"
    def __init__(self, maxv):
        self.maxv = maxv

    def __iter__(self):          <-- 1
        self.a = 0
        self.b = 1
        return self              <-- 2

    def __next__(self):          <-- 3
        fib = self.a
        if fib > self.maxv:
            raise StopIteration  <-- 4
        self.a, self.b = self.b, self.a + self.b
        return fib
```

1. This class is provided with an `__iter__()` method
2. Which returns itself
3. The returned object is provided with a `__next__()` method and thus it is an **iterator**
4. The end of the iteration is signalled with a proper exception

Iterables & iterators -2

Let's make a class wich can be used as an iterable

file: `fibo1.py`

```
class Fibo:
    "Iterator for the Fibonacci series"
    def __init__(self, maxv):
        self.maxv = maxv

    def __iter__(self):          <-- 1
        self.a = 0
        self.b = 1
        return self              <-- 2

    def __next__(self):          <-- 3
        fib = self.a
        if fib > self.maxv:
            raise StopIteration  <-- 4
        self.a, self.b = self.b, self.a + self.b
        return fib
```

1. This class is provided with an `__iter__()` method
2. Which returns itself
3. The returned object is provided with a `__next__()` method and thus it is an **iterator**
4. The end of the iteration is signalled with a proper exception

→

Iterables & iterators -3

Let's see the `Fibo` class in action:

Iterables & iterators -3

Let's see the `Fibo` class in action:

```
>>> from fibo1 import Fibo
>>> for f in Fibo(1000):
...     print(f, end=" ")
...
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 >>>
>>> fb = Fibo(900)
>>> fb
<fibo1.Fibo object at 0x7f669bc0c048>
>>> list(fb)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]
>>>
```

Iterables & iterators -3

Let's see the `Fibo` class in action:

```
>>> from fibo1 import Fibo
>>> for f in Fibo(1000):          <-- 1
...     print(f, end=" ")
...
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 >>>
>>> fb = Fibo(900)               <-- 2
>>> fb
<fibo1.Fibo object at 0x7f669bc0c048>
>>> list(fb)                     <-- 3
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]
>>>
```

Iterables & iterators -3

Let's see the `Fibo` class in action:

```
>>> from fibo1 import Fibo
>>> for f in Fibo(1000):          <-- 1
...     print(f, end=" ")
...
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 >>>
>>> fb = Fibo(900)                <-- 2
>>> fb
<fibo1.Fibo object at 0x7f669bc0c048>
>>> list(fb)                      <-- 3
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]
>>>
```

1. Natural use of iterators is in **for** loops

Iterables & iterators -3

Let's see the `Fibo` class in action:

```
>>> from fibo1 import Fibo
>>> for f in Fibo(1000):          <-- 1
...     print(f, end=" ")
...
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 >>>
>>> fb = Fibo(900)                <-- 2
>>> fb
<fibo1.Fibo object at 0x7f669bc0c048>
>>> list(fb)                      <-- 3
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]
>>>
```

1. Natural use of iterators is in **for** loops
2. `fb` is an instance of class `Fibo`

Iterables & iterators -3

Let's see the `Fibo` class in action:

```
>>> from fibo1 import Fibo
>>> for f in Fibo(1000):          <-- 1
...     print(f, end=" ")
...
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 >>>
>>> fb = Fibo(900)                <-- 2
>>> fb
<fibo1.Fibo object at 0x7f669bc0c048>
>>> list(fb)                      <-- 3
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]
>>>
```

1. Natural use of iterators is in **for** loops
2. `fb` is an instance of class `Fibo`
3. The built-in function `list()` creates a list from an iterator

Iterables & iterators -3

Let's see the `Fibo` class in action:

```
>>> from fibo1 import Fibo
>>> for f in Fibo(1000):          <-- 1
...     print(f, end=" ")
...
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 >>>
>>> fb = Fibo(900)                <-- 2
>>> fb
<fibo1.Fibo object at 0x7f669bc0c048>
>>> list(fb)                      <-- 3
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]
>>>
```

1. Natural use of iterators is in **for** loops
2. `fb` is an instance of class `Fibo`
3. The built-in function `list()` creates a list from an iterator

→

Generators -1

Generators -1

A generator is a syntax construct designed to create simple `iterators`

Generators -1

A generator is a syntax construct designed to create simple `iterators`

file: `fibo2.py`

```
def fibo(maxv):
    a, b = 0, 1
    while a < maxv:
        yield a
        a, b = b, a+b
```

Generators -1

A generator is a syntax construct designed to create simple `iterators`

file: `fibo2.py`

```
def fibo(maxv):          <-- 1
    a, b = 0, 1
    while a < maxv:
        yield a          <-- 2
        a, b = b, a+b
```

Generators -1

A generator is a syntax construct designed to create simple `iterators`

file: `fibo2.py`

```
def fibo(maxv):          <-- 1
    a, b = 0, 1
    while a < maxv:
        yield a          <-- 2
        a, b = b, a+b
```

1. A `generator` looks like a function ...

Generators -1

A generator is a syntax construct designed to create simple `iterators`

file: `fibo2.py`

```
def fibo(maxv):          <-- 1
    a, b = 0, 1
    while a < maxv:
        yield a          <-- 2
        a, b = b, a+b
```

1. A `generator` looks like a function ...
2. ... returning values with the **yield** statement

```
>>> from fibo2 import fibo
>>> for f in fibo(1000):
...     print(f, end=" ")
...
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 >>>
>>> fb=fibo(900)
>>> fb
<generator object fibo at 0x7f5c47901c50>
>>> list(fb)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]
>>>
```

Generators -1

A generator is a syntax construct designed to create simple `iterators`

file: `fibo2.py`

```
def fibo(maxv):          <-- 1
    a, b = 0, 1
    while a < maxv:
        yield a          <-- 2
        a, b = b, a+b
```

1. A `generator` looks like a function ...
2. ... returning values with the **yield** statement

```
>>> from fibo2 import fibo
>>> for f in fibo(1000):          <-- 1
...     print(f, end=" ")
...
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 >>>
>>> fb=fibo(900)          <-- 1
>>> fb
<generator object fibo at 0x7f5c47901c50>
>>> list(fb)          <-- 1
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]
>>>
```

1. A `generator` is used exactly like an `iterator`

Generators -1

A generator is a syntax construct designed to create simple `iterators`

file: `fibo2.py`

```
def fibo(maxv):          <-- 1
    a, b = 0, 1
    while a < maxv:
        yield a          <-- 2
        a, b = b, a+b
```

1. A `generator` looks like a function ...
2. ... returning values with the **yield** statement

```
>>> from fibo2 import fibo
>>> for f in fibo(1000):          <-- 1
...     print(f, end=" ")
...
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 >>>
>>> fb=fibo(900)                  <-- 1
>>> fb
<generator object fibo at 0x7f5c47901c50>
>>> list(fb)                      <-- 1
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]
>>>
```

1. A `generator` is used exactly like an `iterator`

→

Generators -2

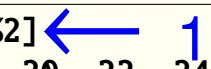
Comprehensions are another form of simple iterators

```
>>> [x for x in range(30) if not x%2]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28]
>>>
```

Generators -2

Comprehensions are another form of simple iterators

```
>>> [x for x in range(30) if not x%2]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28]
>>>
```



1. The built-in function `range(30)` returns an iterator iterating on integers 0..29

Generators -2

Comprehensions are another form of simple iterators

```
>>> [x for x in range(30) if not x%2]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28]
>>>
```

1 ← 1

1. The built-in function `range(30)` returns an iterator iterating on integers 0..29

A `file` object is an `iterable`:

```
>>> fpt = open("file.txt")
>>> for line in fpt:
...   print(line, end="")
...
Linea 1
Linea 2
Ultima linea
>>>
>>> dir(fpt)
['_CHUNK_SIZE', '__class__', ..., '__iter__', ..., '__next__', ...]
```

Generators -2

Comprehensions are another form of simple iterators

```
>>> [x for x in range(30) if not x%2]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28]
>>>
```

1 ← 1

1. The built-in function `range(30)` returns an iterator iterating on integers 0..29

A `file` object is an `iterable`:

```
>>> fpt = open("file.txt")
>>> for line in fpt:
...   print(line, end="")
...
Linea 1
Linea 2
Ultima linea
>>>
>>> dir(fpt)
['_CHUNK_SIZE', '__class__', ..., '__iter__', ..., '__next__', ...]
```

Generators -2

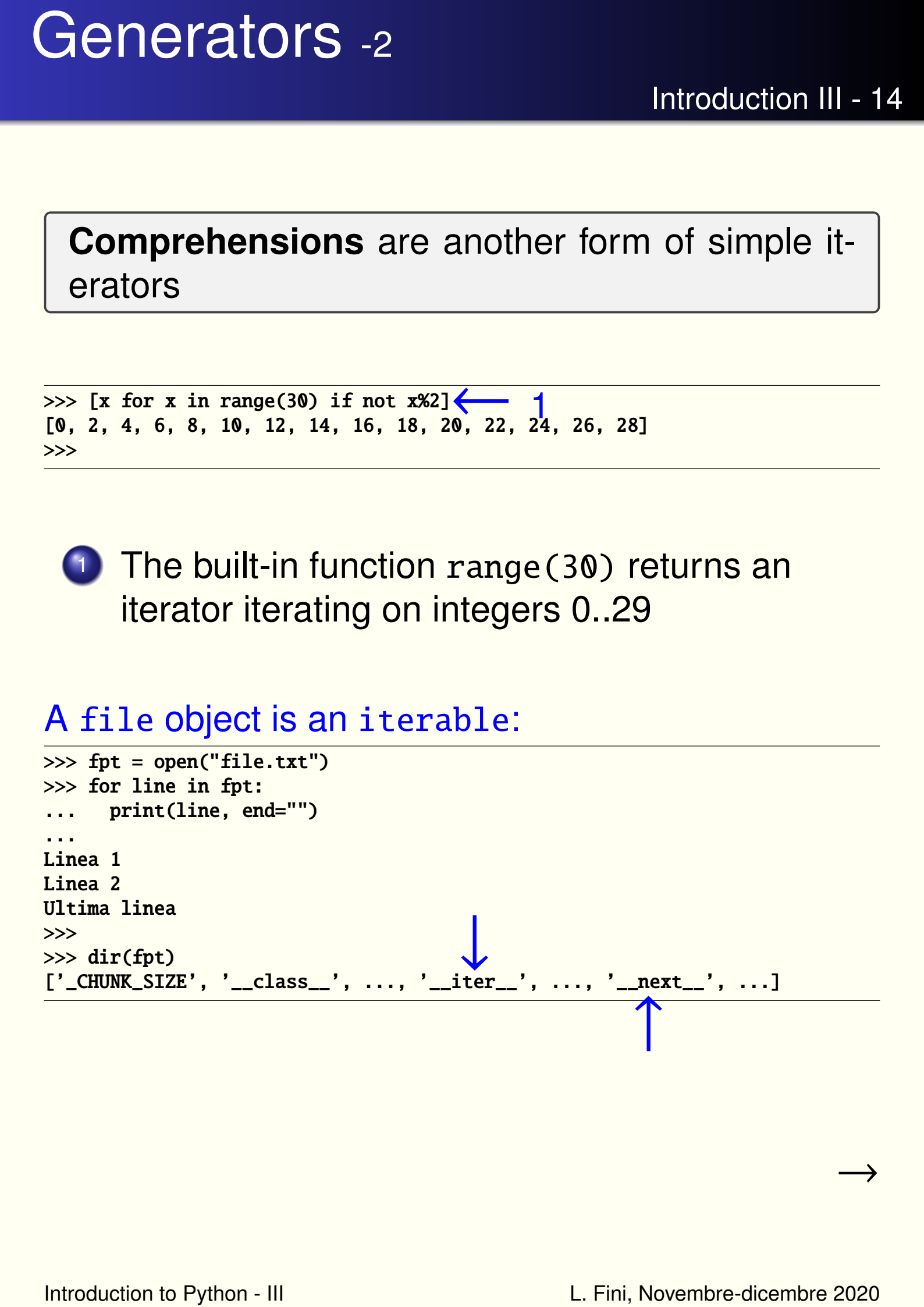
Comprehensions are another form of simple iterators

```
>>> [x for x in range(30) if not x%2]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28]
>>>
```

1. The built-in function `range(30)` returns an iterator iterating on integers 0..29

A `file` object is an `iterable`:

```
>>> fpt = open("file.txt")
>>> for line in fpt:
...   print(line, end="")
...
Linea 1
Linea 2
Ultima linea
>>>
>>> dir(fpt)
['_CHUNK_SIZE', '__class__', ..., '__iter__', ..., '__next__', ...]
```



Context managers -1

Context managers are programming constructs useful whenever your program needs a resource which must to be allocated for use and released after use.

Context managers -1

Context managers are programming constructs useful whenever your program needs a resource which must to be allocated for use and released after use.

Classical file access example:

```
>>> fpt = open("file.txt")
>>> for line in fpt:
...     print(line, end="")
...
Linea 1
Linea 2
Ultima linea
>>> fpt.close()
```

Context managers -1

Context managers are programming constructs useful whenever your program needs a resource which must to be allocated for use and released after use.

Classical file access example:

```
>>> fpt = open("file.txt") ← 1
>>> for line in fpt:
...     print(line, end="")
...
Linea 1
Linea 2
Ultima linea
>>> fpt.close()  ← 2
```

Context managers -1

Context managers are programming constructs useful whenever your program needs a resource which must to be allocated for use and released after use.

Classical file access example:

```
>>> fpt = open("file.txt") ← 1
>>> for line in fpt:
...     print(line, end="")
...
Linea 1
Linea 2
Ultima linea
>>> fpt.close()  ← 2
```

1. In order to read a file you must first **open** it...

Context managers -1

Context managers are programming constructs useful whenever your program needs a resource which must to be allocated for use and released after use.

Classical file access example:

```
>>> fpt = open("file.txt") ← 1
>>> for line in fpt:
...     print(line, end="")
...
Linea 1
Linea 2
Ultima linea
>>> fpt.close()  ← 2
```

1. In order to read a file you must first **open** it...
2. And then **close** it, when you're done

Context managers -1

Context managers are programming constructs useful whenever your program needs a resource which must to be allocated for use and released after use.

Classical file access example:

```
>>> fpt = open("file.txt") ← 1
>>> for line in fpt:
...     print(line, end="")
...
Linea 1
Linea 2
Ultima linea
>>> fpt.close()  ← 2
```

1. In order to read a file you must first **open** it...
2. And then **close** it, when you're done

The same file access "a la Python":

```
>>> with open("file.txt") as fpt:
...     for line in fpt:
...         print(line, end="")
...
```

Context managers -1

Context managers are programming constructs useful whenever your program needs a resource which must to be allocated for use and released after use.

Classical file access example:

```
>>> fpt = open("file.txt") ← 1
>>> for line in fpt:
...     print(line, end="")
...
Linea 1
Linea 2
Ultima linea
>>> fpt.close()  ← 2
```

1. In order to read a file you must first **open** it...
2. And then **close** it, when you're done

The same file access "a la Python":

```
>>> with open("file.txt") as fpt: ← 1,2
...     for line in fpt:
...         print(line, end="")
...  ← 3
```

Context managers -1

Context managers are programming constructs useful whenever your program needs a resource which must to be allocated for use and released after use.

Classical file access example:

```
>>> fpt = open("file.txt") ← 1
>>> for line in fpt:
...     print(line, end="")
...
Linea 1
Linea 2
Ultima linea
>>> fpt.close()  ← 2
```

1. In order to read a file you must first **open** it...
2. And then **close** it, when you're done

The same file access "a la Python":

```
>>> with open("file.txt") as fpt: ← 1,2
...     for line in fpt:
...         print(line, end="")
...  ← 3
```

1. Context managers are used with the **with** statement

Context managers -1

Context managers are programming constructs useful whenever your program needs a resource which must to be allocated for use and released after use.

Classical file access example:

```
>>> fpt = open("file.txt") ← 1
>>> for line in fpt:
...     print(line, end="")
...
Linea 1
Linea 2
Ultima linea
>>> fpt.close()  ← 2
```

1. In order to read a file you must first **open** it...
2. And then **close** it, when you're done

The same file access "a la Python":

```
>>> with open("file.txt") as fpt: ← 1,2
...     for line in fpt:
...         print(line, end="")
...  ← 3
```

1. Context managers are used with the **with** statement
2. A `file` object is also a context manager...

Context managers -1

Context managers are programming constructs useful whenever your program needs a resource which must to be allocated for use and released after use.

Classical file access example:

```
>>> fpt = open("file.txt") ← 1
>>> for line in fpt:
...     print(line, end="")
...
Linea 1
Linea 2
Ultima linea
>>> fpt.close()  ← 2
```

1. In order to read a file you must first **open** it...
2. And then **close** it, when you're done

The same file access "a la Python":

```
>>> with open("file.txt") as fpt: ← 1,2
...     for line in fpt:
...         print(line, end="")
...  ← 3
```

1. Context managers are used with the **with** statement
2. A `file` object is also a context manager...
3. You need not to bother closing the file, because the context manager does it for you: even in case of errors!

Context managers -1

Context managers are programming constructs useful whenever your program needs a resource which must to be allocated for use and released after use.

Classical file access example:

```
>>> fpt = open("file.txt") ← 1
>>> for line in fpt:
...     print(line, end="")
...
Linea 1
Linea 2
Ultima linea
>>> fpt.close()  ← 2
```

1. In order to read a file you must first **open** it...
2. And then **close** it, when you're done

The same file access "a la Python":

```
>>> with open("file.txt") as fpt: ← 1,2
...     for line in fpt:
...         print(line, end="")
...  ← 3
```

1. Context managers are used with the **with** statement
2. A `file` object is also a context manager...
3. You need not to bother closing the file, because the context manager does it for you: even in case of errors!

Context managers -2

You can build your own context manager by creating a class as in the following example:

Example: a `File` object

```
class File():
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode

    def __enter__(self):
        self.open_file = open(self.filename, self.mode)
        return self.open_file

    def __exit__(self, *args):
        self.open_file.close()
```

Context managers -2

You can build your own context manager by creating a class as in the following example:

Example: a `File` object

```
class File():
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode

    def __enter__(self):          <-- 1
        self.open_file = open(self.filename, self.mode)
        return self.open_file

    def __exit__(self, *args):    <-- 2
        self.open_file.close()
```

Context managers -2

You can build your own context manager by creating a class as in the following example:

Example: a `File` object

```
class File():
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode

    def __enter__(self):                    <-- 1
        self.open_file = open(self.filename, self.mode)
        return self.open_file

    def __exit__(self, *args):              <-- 2
        self.open_file.close()
```

1. Context managers must define the method **`__enter__()`** ...

Context managers -2

You can build your own context manager by creating a class as in the following example:

Example: a `File` object

```
class File():
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode

    def __enter__(self):          <-- 1
        self.open_file = open(self.filename, self.mode)
        return self.open_file

    def __exit__(self, *args):    <-- 2
        self.open_file.close()
```

1. Context managers must define the method **`__enter__()`** ...
2. ... and the method **`__exit__()`**

Context managers -2

You can build your own context manager by creating a class as in the following example:

Example: a `File` object

```
class File():
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode

    def __enter__(self):                    <-- 1
        self.open_file = open(self.filename, self.mode)
        return self.open_file

    def __exit__(self, *args):              <-- 2
        self.open_file.close()
```

1. Context managers must define the method **`__enter__()`** ...
2. ... and the method **`__exit__()`**

Context managers are so useful in everyday's programming that Python provides helpers for building context managers in a dedicated package: **`contextlib`**

Context managers -2

You can build your own context manager by creating a class as in the following example:

Example: a `File` object

```
class File():
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode

    def __enter__(self):                    <-- 1
        self.open_file = open(self.filename, self.mode)
        return self.open_file

    def __exit__(self, *args):              <-- 2
        self.open_file.close()
```

1. Context managers must define the method **`__enter__()`** ...
2. ... and the method **`__exit__()`**

Context managers are so useful in everyday's programming that Python provides helpers for building context managers in a dedicated package: **`contextlib`**

→

Decorators -1

Decorators -1

Decorators are language constructs which allow to dynamically alter a function, a method or a class without modifying the code of the function or using a subclass.

Decorators -1

Decorators are language constructs which allow to dynamically alter a function, a method or a class without modifying the code of the function or using a subclass.

file: decorator.py

```
import time

def my_timer(f):
    def wrapper(*pw, **kw):
        tm0 = time.time()
        ret = f(*pw, **kw)
        tm1 = time.time()
        print("Elapsed time:", tm1-tm0, "s")
    return wrapper
```

Decorators -1

Decorators are language constructs which allow to dynamically alter a function, a method or a class without modifying the code of the function or using a subclass.

file: decorator.py

```
import time

def my_timer(f):                                    ← 1
    def wrapper(*pw, **kw):
        tm0 = time.time()
        ret = f(*pw, **kw)                          ← 3
        tm1 = time.time()
        print("Elapsed time:", tm1-tm0, "s")        ← 3
    return wrapper                                  ← 2
```

Decorators -1

Decorators are language constructs which allow to dynamically alter a function, a method or a class without modifying the code of the function or using a subclass.

file: decorator.py

```
import time

def my_timer(f):                                    ← 1
    def wrapper(*pw, **kw):
        tm0 = time.time()
        ret = f(*pw, **kw)                          ← 3
        tm1 = time.time()
        print("Elapsed time:", tm1-tm0, "s")        ← 3
    return wrapper                                  ← 2
```

1. A function decorator is a function accepting a function as only argument ...

Decorators -1

Decorators are language constructs which allow to dynamically alter a function, a method or a class without modifying the code of the function or using a subclass.

file: decorator.py

```
import time

def my_timer(f):                                    ← 1
    def wrapper(*pw, **kw):
        tm0 = time.time()
        ret = f(*pw, **kw)                          ← 3
        tm1 = time.time()
        print("Elapsed time:", tm1-tm0, "s")        ← 3
    return wrapper                                  ← 2
```

1. A function decorator is a function accepting a function as only argument ...
2. ... and returning a function

Decorators -1

Decorators are language constructs which allow to dynamically alter a function, a method or a class without modifying the code of the function or using a subclass.

file: `decorator.py`

```
import time

def my_timer(f):                                    <- 1
    def wrapper(*pw, **kw):
        tm0 = time.time()
        ret = f(*pw, **kw)                          <- 3
        tm1 = time.time()
        print("Elapsed time:", tm1-tm0, "s")        <- 3
    return wrapper                                  <- 2
```

1. A function decorator is a function accepting a function as only argument ...
2. ... and returning a function
3. The returned function does “something” with the original one. In this case:
   - gets current time
   - calls the original function
   - computes and prints the elapsed time

Decorators -1

Decorators are language constructs which allow to dynamically alter a function, a method or a class without modifying the code of the function or using a subclass.

file: `decorator.py`

```
import time

def my_timer(f):                                    ← 1
    def wrapper(*pw, **kw):
        tm0 = time.time()
        ret = f(*pw, **kw)                          ← 3
        tm1 = time.time()
        print("Elapsed time:", tm1-tm0, "s")        ← 3
    return wrapper                                  ← 2
```

1. A function decorator is a function accepting a function as only argument ...
2. ... and returning a function
3. The returned function does “something” with the original one. In this case:
   - gets current time
   - calls the original function
   - computes and prints the elapsed time

→

Decorators -2

Example: decorating a function - File: fibo3.py

```
from decorator import my_timer

@my_timer
def fibo_print(n):
    "Prints n elements of the Fibonacci series"
    result = []
    a, b = 0, 1
    while len(result) < n:
        result.append(b)
        a, b = b, a+b
        print(a, end=" ")
    print()
```

Decorators -2

Example: decorating a function - File: fibo3.py

```
from decorator import my_timer

@my_timer ← 1
def fibo_print(n):
    "Prints n elements of the Fibonacci series"
    result = []
    a, b = 0, 1
    while len(result) < n:
        result.append(b)
        a, b = b, a+b
        print(a, end=" ")
    print()
```

Decorators -2

Example: decorating a function - File: fibo3.py

```
from decorator import my_timer

@my_timer ← 1
def fibo_print(n):
    "Prints n elements of the Fibonacci series"
    result = []
    a, b = 0, 1
    while len(result) < n:
        result.append(b)
        a, b = b, a+b
        print(a, end=" ")
    print()
```

1. The @ operator “decorates” the following function with the given decorator

Decorators -2

Example: decorating a function - File: fibo3.py

```
from decorator import my_timer

@my_timer ← 1
def fibo_print(n):
    "Prints n elements of the Fibonacci series"
    result = []
    a, b = 0, 1
    while len(result) < n:
        result.append(b)
        a, b = b, a+b
        print(a, end=" ")
    print()
```

1. The @ operator “decorates” the following function with the given decorator

At execution:

```
>>> from fibo3 import fibo_print
>>> fibo_print(8)
1 1 2 3 5 8 13 21
Elapsed time: 4.1961669921875e-05 s
>>>
```

Decorators -2

Example: decorating a function - File: fibo3.py

```
from decorator import my_timer

@my_timer ⟵ 1
def fibo_print(n):
    "Prints n elements of the Fibonacci series"
    result = []
    a, b = 0, 1
    while len(result) < n:
        result.append(b)
        a, b = b, a+b
        print(a, end=" ")
    print()
```

1. The @ operator “decorates” the following function with the given decorator

At execution:

```
>>> from fibo3 import fibo_print
>>> fibo_print(8)
1 1 2 3 5 8 13 21 ⟵ 1
Elapsed time: 4.1961669921875e-05 s ⟵ 2
>>>
```

Decorators -2

Example: decorating a function - File: fibo3.py

```
from decorator import my_timer

@my_timer ⟵ 1
def fibo_print(n):
    "Prints n elements of the Fibonacci series"
    result = []
    a, b = 0, 1
    while len(result) < n:
        result.append(b)
        a, b = b, a+b
        print(a, end=" ")
    print()
```

1. The @ operator “decorates” the following function with the given decorator

At execution:

```
>>> from fibo3 import fibo_print
>>> fibo_print(8)
1 1 2 3 5 8 13 21 ⟵ 1
Elapsed time: 4.1961669921875e-05 s ⟵ 2
>>>
```

1. The original function does its job

Decorators -2

Example: decorating a function - File: fibo3.py

```
from decorator import my_timer

@my_timer ⟵ 1
def fibo_print(n):
    "Prints n elements of the Fibonacci series"
    result = []
    a, b = 0, 1
    while len(result) < n:
        result.append(b)
        a, b = b, a+b
        print(a, end=" ")
    print()
```

1. The @ operator “decorates” the following function with the given decorator

At execution:

```
>>> from fibo3 import fibo_print
>>> fibo_print(8)
1 1 2 3 5 8 13 21 ⟵ 1
Elapsed time: 4.1961669921875e-05 s ⟵ 2
>>>
```

1. The original function does its job
2. The decorator does its job

Decorators -2

Example: decorating a function - File: fibo3.py

```
from decorator import my_timer

@my_timer ⟵ 1
def fibo_print(n):
    "Prints n elements of the Fibonacci series"
    result = []
    a, b = 0, 1
    while len(result) < n:
        result.append(b)
        a, b = b, a+b
        print(a, end=" ")
    print()
```

1. The @ operator “decorates” the following function with the given decorator

At execution:

```
>>> from fibo3 import fibo_print
>>> fibo_print(8)
1 1 2 3 5 8 13 21 ⟵ 1
Elapsed time: 4.1961669921875e-05 s ⟵ 2
>>>
```

1. The original function does its job
2. The decorator does its job

→

Development process

An in-deep coverage of development and test techniques is far beyond the limits of this course, but a few notes and suggestions may be useful for the interested audience.

Development process

An in-deep coverage of development and test techniques is far beyond the limits of this course, but a few notes and suggestions may be useful for the interested audience.

We will examine some tools useful for the various developing phases:

- Coding
- Code static check
- Debugging
- Testing

Development process

An in-deep coverage of development and test techniques is far beyond the limits of this course, but a few notes and suggestions may be useful for the interested audience.

We will examine some tools useful for the various developing phases:

- Coding
- Code static check
- Debugging
- Testing

There are also some Integrated Development Environment (IDE) programs, both commercial and freely available, which provide several development tools in a single application.

Development process

An in-deep coverage of development and test techniques is far beyond the limits of this course, but a few notes and suggestions may be useful for the interested audience.

We will examine some tools useful for the various developing phases:

- Coding
- Code static check
- Debugging
- Testing

There are also some Integrated Development Environment (IDE) programs, both commercial and freely available, which provide several development tools in a single application.

→

Language aware editors

Language aware editors

- Language aware editors are text editors which highlight the various elements of the programming language by using different colors

Language aware editors

- Language aware editors are text editors which highlight the various elements of the programming language by using different colors
- The programmer can thus better distinguish reserved words, variables, constants and so on

Language aware editors

- Language aware editors are text editors which highlight the various elements of the programming language by using different colors
- The programmer can thus better distinguish reserved words, variables, constants and so on

Examples:

- `vim`
- `emacs`
- `nano`

Language aware editors

- Language aware editors are text editors which highlight the various elements of the programming language by using different colors
- The programmer can thus better distinguish reserved words, variables, constants and so on

Examples:

- `vim`
- `emacs`
- `nano`

→

Static analysis tools -1

Dynamic programming languages such as Python have very limited early diagnostic capabilities: most errors are discovered only when (that particular portion of) the program is executed.

Static analysis tools -1

Dynamic programming languages such as Python have very limited early diagnostic capabilities: most errors are discovered only when (that particular portion of) the program is executed.

File: bugged.py

```
import time

a = 502
i = 0
while a > 0:
    time.sleep(1)
    print(a)
    if a == 500:
        i = j+1
        print("500 seconds to the end")
    a -= 1
```

Static analysis tools -1

Dynamic programming languages such as Python have very limited early diagnostic capabilities: most errors are discovered only when (that particular portion of) the program is executed.

File: bugged.py

```
import time

a = 502
i = 0
while a > 0:
    time.sleep(1)
    print(a)
    if a == 500:
        i = j+1
        print("500 seconds to the end")
    a -= 1
```

Static analysis tools -1

Dynamic programming languages such as Python have very limited early diagnostic capabilities: most errors are discovered only when (that particular portion of) the program is executed.

File: bugged.py

```
import time

a = 502
i = 0
while a > 0:
    time.sleep(1)
    print(a)
    if a == 500:
        i = j+1
        print("500 seconds to the end")
    a -= 1
```

the bug is here

When running the program:

```
$ python bugged.py
502
501
500
Traceback (most recent call last):
  File "bugged.py", line 9, in <module>
    i = j+1
NameError: name 'j' is not defined
```

Static analysis tools -1

Dynamic programming languages such as Python have very limited early diagnostic capabilities: most errors are discovered only when (that particular portion of) the program is executed.

File: bugged.py

```
import time

a = 502
i = 0
while a > 0:
    time.sleep(1)
    print(a)
    if a == 500:
        i = j+1
        print("500 seconds to the end")
    a -= 1
```

← the bug is here

When running the program:

```
$ python bugged.py
502
501
500
Traceback (most recent call last):
  File "bugged.py", line 9, in <module>
    i = j+1
NameError: name 'j' is not defined
```

← 1,2

1. The error has been found after three iterations

Static analysis tools -1

Dynamic programming languages such as Python have very limited early diagnostic capabilities: most errors are discovered only when (that particular portion of) the program is executed.

File: bugged.py

```
import time

a = 502
i = 0
while a > 0:
    time.sleep(1)
    print(a)
    if a == 500:
        i = j+1
        print("500 seconds to the end")
    a -= 1
```

← the bug is here

When running the program:

```
$ python bugged.py
502
501
500
Traceback (most recent call last):
  File "bugged.py", line 9, in <module>
    i = j+1
NameError: name 'j' is not defined
```

← 1,2

1. The error has been found after three iterations
2. But if the value of **a** were, say, 10000, the error whould have happened after 9500 iterations!

Static analysis tools -1

Dynamic programming languages such as Python have very limited early diagnostic capabilities: most errors are discovered only when (that particular portion of) the program is executed.

File: bugged.py

```
import time

a = 502
i = 0
while a > 0:
    time.sleep(1)
    print(a)
    if a == 500:
        i = j+1
        print("500 seconds to the end")
    a -= 1
```

← the bug is here

When running the program:

```
$ python bugged.py
502
501
500
Traceback (most recent call last):
  File "bugged.py", line 9, in <module>
    i = j+1
NameError: name 'j' is not defined
```

← 1,2

1. The error has been found after three iterations
2. But if the value of **a** were, say, 10000, the error whould have happened after 9500 iterations!

→

Static analysis tools -2

pylint is a static code analizer which can discover many bugs before the program is actually executed.

Static analysis tools -2

pylint is a static code analizer which can discover many bugs before the program is actually executed.

```
$ pylint bugged.py
************** Module bugged
C:  1, 0: Missing module docstring (missing-docstring)
C:  3, 0: Constant name "a" doesn't conform to UPPER_CASE naming style (invalid-n
E:  9,12: Undefined variable 'j' (undefined-variable)

-----------------------------------
Your code has been rated at 3.00/10
```

Static analysis tools -2

pylint is a static code analizer which can discover many bugs before the program is actually executed.

```
$ pylint bugged.py   <-- 1
************** Module bugged
C:  1, 0: Missing module docstring (missing-docstring) <-- 3
C:  3, 0: Constant name "a" doesn't conform to UPPER_CASE naming style (invalid-n
E:  9,12: Undefined variable 'j' (undefined-variable) <-- 2

-----------------------------------
Your code has been rated at 3.00/10 <-- 4
```

Static analysis tools -2

pylint is a static code analizer which can discover many bugs before the program is actually executed.

```
$ pylint bugged.py   <-- 1
************** Module bugged
C:  1, 0: Missing module docstring (missing-docstring) <-- 3
C:  3, 0: Constant name "a" doesn't conform to UPPER_CASE naming style (invalid-r
E:  9,12: Undefined variable 'j' (undefined-variable) <-- 2

-----------------------------------
Your code has been rated at 3.00/10 <-- 4
```

1. Using **pylint** to analyze a Python file

Static analysis tools -2

pylint is a static code analizer which can discover many bugs before the program is actually executed.

```
$ pylint bugged.py   <- 1
************** Module bugged
C:  1, 0: Missing module docstring (missing-docstring) <- 3
C:  3, 0: Constant name "a" doesn't conform to UPPER_CASE naming style (invalid-n
E:  9,12: Undefined variable 'j' (undefined-variable) <- 2

-----------------------------------
Your code has been rated at 3.00/10 <- 4
```

1. Using **pylint** to analyze a Python file
2. The error is detected by `pylint`

Static analysis tools -2

pylint is a static code analizer which can discover many bugs before the program is actually executed.

```
$ pylint bugged.py                                                    <- 1
************** Module bugged
C:  1, 0: Missing module docstring (missing-docstring)                <- 3
C:  3, 0: Constant name "a" doesn't conform to UPPER_CASE naming style (invalid-n
E:  9,12: Undefined variable 'j' (undefined-variable)                 <- 2

-----------------------------------
Your code has been rated at 3.00/10                                   <- 4
```

1. Using **pylint** to analyze a Python file
2. The error is detected by `pylint`
3. `pylint` performs also several "quality checks"

Static analysis tools -2

pylint is a static code analizer which can discover many bugs before the program is actually executed.

```
$ pylint bugged.py                                                    <- 1
************** Module bugged
C:  1, 0: Missing module docstring (missing-docstring)                <- 3
C:  3, 0: Constant name "a" doesn't conform to UPPER_CASE naming style (invalid-n
E:  9,12: Undefined variable 'j' (undefined-variable)                 <- 2

-----------------------------------
Your code has been rated at 3.00/10                                   <- 4
```

1. Using **pylint** to analyze a Python file
2. The error is detected by `pylint`
3. `pylint` performs also several "quality checks"
4. And computes a global quality score.

Static analysis tools -2

pylint is a static code analizer which can discover many bugs before the program is actually executed.

```
$ pylint bugged.py                                                    <- 1
************** Module bugged
C:  1, 0: Missing module docstring (missing-docstring)                <- 3
C:  3, 0: Constant name "a" doesn't conform to UPPER_CASE naming style (invalid-n
E:  9,12: Undefined variable 'j' (undefined-variable)                 <- 2

-----------------------------------
Your code has been rated at 3.00/10                                   <- 4
```

1. Using **pylint** to analyze a Python file
2. The error is detected by `pylint`
3. `pylint` performs also several "quality checks"
4. And computes a global quality score.

→

Static analysis tools -3

mypy uses Python 3 function syntax to perform static type checking on your program.

Static analysis tools -3

mypy uses Python 3 function syntax to perform static type checking on your program.

File: fibo4.py

```
def fibo(maxv: int):
    a, b = 0, 1
    while a < maxv:
        yield a
        a, b = b, a+b

if __name__ == "__main__":
    for f in fibo(300.5):
        print(f, end=" ")
    print()
```

Static analysis tools -3

mypy uses Python 3 function syntax to perform static type checking on your program.

File: fibo4.py

```
def fibo(maxv: int):
    a, b = 0, 1
    while a < maxv:
        yield a
        a, b = b, a+b

if __name__ == "__main__":
    for f in fibo(300.5):
        print(f, end=" ")
    print()
```

← annotation

Static analysis tools -3

mypy uses Python 3 function syntax to perform static type checking on your program.

File: fibo4.py

```
def fibo(maxv: int):          <-- annotation
    a, b = 0, 1
    while a < maxv:
        yield a
        a, b = b, a+b

if __name__ == "__main__":
    for f in fibo(300.5):
        print(f, end=" ")
    print()
```

```
$ python fibo4.py          <-- 1
0 1 1 2 3 5 8 13 21 34 55 89 144 233
$ mypy fibo4.py            <-- 2
fibo4.py:8: error: Argument 1 to "fibo" has incompatible type "float"; expected
```

Static analysis tools -3

mypy uses Python 3 function syntax to perform static type checking on your program.

File: fibo4.py

```
def fibo(maxv: int):          <-- annotation
    a, b = 0, 1
    while a < maxv:
        yield a
        a, b = b, a+b

if __name__ == "__main__":
    for f in fibo(300.5):
        print(f, end=" ")
    print()
```

```
$ python fibo4.py          <-- 1
0 1 1 2 3 5 8 13 21 34 55 89 144 233
$ mypy fibo4.py            <-- 2
fibo4.py:8: error: Argument 1 to "fibo" has incompatible type "float"; expected
```

1. Annotations are ignored by the Python interpreter

Static analysis tools -3

mypy uses Python 3 function syntax to perform static type checking on your program.

File: fibo4.py

```
def fibo(maxv: int):          <-- annotation
    a, b = 0, 1
    while a < maxv:
        yield a
        a, b = b, a+b

if __name__ == "__main__":
    for f in fibo(300.5):
        print(f, end=" ")
    print()
```

```
$ python fibo4.py          <-- 1
0 1 1 2 3 5 8 13 21 34 55 89 144 233
$ mypy fibo4.py            <-- 2
fibo4.py:8: error: Argument 1 to "fibo" has incompatible type "float"; expected
```

1. Annotations are ignored by the Python interpreter
2. But are used by `mypy` to do type checking

The standard package **typing** provides more powerful tools to improve static type checking

Static analysis tools -3

mypy uses Python 3 function syntax to perform static type checking on your program.

File: fibo4.py

```
def fibo(maxv: int):          ← annotation
    a, b = 0, 1
    while a < maxv:
        yield a
        a, b = b, a+b

if __name__ == "__main__":
    for f in fibo(300.5):
        print(f, end=" ")
    print()
```

```
$ python fibo4.py          ← 1
0 1 1 2 3 5 8 13 21 34 55 89 144 233
$ mypy fibo4.py            ← 2
fibo4.py:8: error: Argument 1 to "fibo" has incompatible type "float"; expected
```

1. Annotations are ignored by the Python interpreter
2. But are used by `mypy` to do type checking

The standard package **typing** provides more powerful tools to improve static type checking

→

The python debugger-1

Python is provided with an interactive debugger (**pdb**) which allows to control program execution

The python debugger-1

Python is provided with an interactive debugger (**pdb**) which allows to control program execution

With **pdb** you can:

The python debugger-1

Python is provided with an interactive debugger (**pdb**) which allows to control program execution

With **pdb** you can:

- Define breakpoints

The python debugger-1

Python is provided with an interactive debugger (**pdb**) which allows to control program execution

With **pdb** you can:

- Define breakpoints
- Execute a program step by step

The python debugger-1

Python is provided with an interactive debugger (**pdb**) which allows to control program execution

With **pdb** you can:

- Define breakpoints
- Execute a program step by step
- Look into variables

The python debugger-1

Python is provided with an interactive debugger (**pdb**) which allows to control program execution

With **pdb** you can:

- Define breakpoints
- Execute a program step by step
- Look into variables
- Execute functions at the prompt

The python debugger-1

Python is provided with an interactive debugger (**pdb**) which allows to control program execution

With **pdb** you can:

- Define breakpoints
- Execute a program step by step
- Look into variables
- Execute functions at the prompt

→

The python debugger-1

Let's see a brief example:

```
 pdb fibo.py
> .../code/fibo.py(1)<module>()
-> "Module for the computation of Fibonacci series"
(Pdb) b 9
Breakpoint 1 at .../code/fibo.py:9
(Pdb) c
> .../code/fibo.py(9)fibo()
-> a, b = 0, 1
(Pdb) n
> .../code/fibo.py(10)fibo()
-> while b < n:
  ...
(Pdb) p result
[1, 1, 2]
```

The python debugger-1

Let's see a brief example:

```
 pdb fibo.py                                  <-- 1
> .../code/fibo.py(1)<module>()
-> "Module for the computation of Fibonacci series"
(Pdb) b 9                                     <-- 2,3
Breakpoint 1 at .../code/fibo.py:9
(Pdb) c                                       <-- 4
> .../code/fibo.py(9)fibo()
-> a, b = 0, 1                                <-- 5,6
(Pdb) n                                       <-- 6
> .../code/fibo.py(10)fibo()
-> while b < n:
  ...
(Pdb) p result                                <-- 7
[1, 1, 2]
```

The python debugger-1

Let's see a brief example:

```
 pdb fibo.py                    <-- 1
> .../code/fibo.py(1)<module>()
-> "Module for the computation of Fibonacci series"
(Pdb) b 9                       <-- 2,3
Breakpoint 1 at .../code/fibo.py:9
(Pdb) c                         <-- 4
> .../code/fibo.py(9)fibo()
-> a, b = 0, 1                  <-- 5,6
(Pdb) n                         <-- 6
> .../code/fibo.py(10)fibo()
-> while b < n:
  ...
(Pdb) p result                  <-- 7
[1, 1, 2]
```

1. Start the debugger on program `fibo.py`.

The python debugger-1

Let's see a brief example:

```
 pdb fibo.py                    ← 1
> .../code/fibo.py(1)<module>()
-> "Module for the computation of Fibonacci series"
(Pdb) b 9                       ← 2,3
Breakpoint 1 at .../code/fibo.py:9
(Pdb) c                         ← 4
> .../code/fibo.py(9)fibo()
-> a, b = 0, 1                  ← 5,6
(Pdb) n                         ← 6
> .../code/fibo.py(10)fibo()
-> while b < n:
  ...
(Pdb) p result                  ← 7
[1, 1, 2]
```

1. Start the debugger on program `fibo.py`.
2. The debugger stops immediately at the first line of the program after showing the line of code just to be executed

The python debugger-1

Let's see a brief example:

```
 pdb fibo.py                    <-- 1
> .../code/fibo.py(1)<module>()
-> "Module for the computation of Fibonacci series"
(Pdb) b 9                       <-- 2,3
Breakpoint 1 at .../code/fibo.py:9
(Pdb) c                         <-- 4
> .../code/fibo.py(9)fibo()
-> a, b = 0, 1                  <-- 5,6
(Pdb) n                         <-- 6
> .../code/fibo.py(10)fibo()
-> while b < n:
  ...
(Pdb) p result                  <-- 7
[1, 1, 2]
```

1. Start the debugger on program `fibo.py`.
2. The debugger stops immediately at the first line of the program after showing the line of code just to be executed
3. Set a breakpoint at line number 9

The python debugger-1

Let's see a brief example:

```
 pdb fibo.py  <-- 1
> .../code/fibo.py(1)<module>()
-> "Module for the computation of Fibonacci series"
(Pdb) b 9  <-- 2,3
Breakpoint 1 at .../code/fibo.py:9
(Pdb) c  <-- 4
> .../code/fibo.py(9)fibo()
-> a, b = 0, 1  <-- 5,6
(Pdb) n  <-- 6
> .../code/fibo.py(10)fibo()
-> while b < n:
  ...
(Pdb) p result  <-- 7
[1, 1, 2]
```

1. Start the debugger on program `fibo.py`.
2. The debugger stops immediately at the first line of the program after showing the line of code just to be executed
3. Set a breakpoint at line number 9
4. Continue (actually: start) the execution

The python debugger-1

Let's see a brief example:

```
 pdb fibo.py  <-- 1
> .../code/fibo.py(1)<module>()
-> "Module for the computation of Fibonacci series"
(Pdb) b 9  <-- 2,3
Breakpoint 1 at .../code/fibo.py:9
(Pdb) c  <-- 4
> .../code/fibo.py(9)fibo()
-> a, b = 0, 1  <-- 5,6
(Pdb) n  <-- 6
> .../code/fibo.py(10)fibo()
-> while b < n:
  ...
(Pdb) p result  <-- 7
[1, 1, 2]
```

1. Start the debugger on program `fibo.py`.
2. The debugger stops immediately at the first line of the program after showing the line of code just to be executed
3. Set a breakpoint at line number 9
4. Continue (actually: start) the execution
5. The debugger stops at the breakpoint (just before executing line 9)

The python debugger-1

Let's see a brief example:

```
 pdb fibo.py  <-- 1
> .../code/fibo.py(1)<module>()
-> "Module for the computation of Fibonacci series"
(Pdb) b 9  <-- 2,3
Breakpoint 1 at .../code/fibo.py:9
(Pdb) c  <-- 4
> .../code/fibo.py(9)fibo()
-> a, b = 0, 1  <-- 5,6
(Pdb) n  <-- 6
> .../code/fibo.py(10)fibo()
-> while b < n:
  ...
(Pdb) p result  <-- 7
[1, 1, 2]
```

1. Start the debugger on program `fibo.py`.
2. The debugger stops immediately at the first line of the program after showing the line of code just to be executed
3. Set a breakpoint at line number 9
4. Continue (actually: start) the execution
5. The debugger stops at the breakpoint (just before executing line 9)
6. Execute a number of steps (9)

The python debugger-1

Let's see a brief example:

```
 pdb fibo.py  <-- 1
> .../code/fibo.py(1)<module>()
-> "Module for the computation of Fibonacci series"
(Pdb) b 9  <-- 2,3
Breakpoint 1 at .../code/fibo.py:9
(Pdb) c  <-- 4
> .../code/fibo.py(9)fibo()
-> a, b = 0, 1  <-- 5,6
(Pdb) n  <-- 6
> .../code/fibo.py(10)fibo()
-> while b < n:
  ...
(Pdb) p result  <-- 7
[1, 1, 2]
```

1. Start the debugger on program `fibo.py`.
2. The debugger stops immediately at the first line of the program after showing the line of code just to be executed
3. Set a breakpoint at line number 9
4. Continue (actually: start) the execution
5. The debugger stops at the breakpoint (just before executing line 9)
6. Execute a number of steps (9)
7. Now see what's in variable `result`

The python debugger-1

Let's see a brief example:

```
 pdb fibo.py  <-- 1
> .../code/fibo.py(1)<module>()
-> "Module for the computation of Fibonacci series"
(Pdb) b 9  <-- 2,3
Breakpoint 1 at .../code/fibo.py:9
(Pdb) c  <-- 4
> .../code/fibo.py(9)fibo()
-> a, b = 0, 1  <-- 5,6
(Pdb) n  <-- 6
> .../code/fibo.py(10)fibo()
-> while b < n:
  ...
(Pdb) p result  <-- 7
[1, 1, 2]
```

1. Start the debugger on program `fibo.py`.
2. The debugger stops immediately at the first line of the program after showing the line of code just to be executed
3. Set a breakpoint at line number 9
4. Continue (actually: start) the execution
5. The debugger stops at the breakpoint (just before executing line 9)
6. Execute a number of steps (9)
7. Now see what's in variable `result`

→

Testing tools -1

Python provides a package to support the generation of testing procedures for python code:

unittest

Testing tools -1

Python provides a package to support the generation of testing procedures for python code:

unittest

file: `test_number.py` - A simple example

```
import unittest
from numberplus1 import NumberPlus

class TestNumber(unittest.TestCase):

    def test_upper(self):
        self.assertTrue(NumberPlus(3).upper(), "Three")

    def test_sum(self):
        self.assertEqual((NumberPlus(2)+NumberPlus(5)).value, 7)

    def test_overflow(self):
        with self.assertRaises(IndexError):
            NumberPlus(12)

if __name__ == "__main__":
    unittest.main()
```

Testing tools -1

Python provides a package to support the generation of testing procedures for python code:

unittest

file: `test_number.py` - A simple example

```
import unittest
from numberplus1 import NumberPlus

class TestNumber(unittest.TestCase):          <-- 1

    def test_upper(self):                     <-- 2
        self.assertTrue(NumberPlus(3).upper(), "Three")

    def test_sum(self):                       <-- 2
        self.assertEqual((NumberPlus(2)+NumberPlus(5)).value, 7)

    def test_overflow(self):                  <-- 2
        with self.assertRaises(IndexError):
            NumberPlus(12)

if __name__ == "__main__":
    unittest.main()                           <-- 3
```

Testing tools -1

Python provides a package to support the generation of testing procedures for python code:

unittest

file: `test_number.py` - A simple example

```
import unittest
from numberplus1 import NumberPlus

class TestNumber(unittest.TestCase):          <-- 1

    def test_upper(self):                     <-- 2
        self.assertTrue(NumberPlus(3).upper(), "Three")

    def test_sum(self):                       <-- 2
        self.assertEqual((NumberPlus(2)+NumberPlus(5)).value, 7)

    def test_overflow(self):                  <-- 2
        with self.assertRaises(IndexError):
            NumberPlus(12)

if __name__ == "__main__":
    unittest.main()                           <-- 3
```

1. A “test case” is a class derived from `TestCase`

Testing tools -1

Python provides a package to support the generation of testing procedures for python code:

unittest

file: `test_number.py` - A simple example

```
import unittest
from numberplus1 import NumberPlus

class TestNumber(unittest.TestCase):          <-- 1

    def test_upper(self):                     <-- 2
        self.assertTrue(NumberPlus(3).upper(), "Three")

    def test_sum(self):                       <-- 2
        self.assertEqual((NumberPlus(2)+NumberPlus(5)).value, 7)

    def test_overflow(self):                  <-- 2
        with self.assertRaises(IndexError):
            NumberPlus(12)

if __name__ == "__main__":
    unittest.main()                           <-- 3
```

1. A “test case” is a class derived from `TestCase`
2. The test case is made up of a set of tests (methods named `test_...`)

Testing tools -1

Python provides a package to support the generation of testing procedures for python code:

unittest

file: `test_number.py` - A simple example

```
import unittest
from numberplus1 import NumberPlus

class TestNumber(unittest.TestCase):          <-- 1

    def test_upper(self):                     <-- 2
        self.assertTrue(NumberPlus(3).upper(), "Three")

    def test_sum(self):                       <-- 2
        self.assertEqual((NumberPlus(2)+NumberPlus(5)).value, 7)

    def test_overflow(self):                  <-- 2
        with self.assertRaises(IndexError):
            NumberPlus(12)

if __name__ == "__main__":
    unittest.main()                           <-- 3
```

1. A “test case” is a class derived from `TestCase`
2. The test case is made up of a set of tests (methods named `test_...`)
3. Here’s where the test is run

Testing tools -1

Python provides a package to support the generation of testing procedures for python code:

unittest

file: `test_number.py` - A simple example

```
import unittest
from numberplus1 import NumberPlus

class TestNumber(unittest.TestCase):          <-- 1

    def test_upper(self):                     <-- 2
        self.assertTrue(NumberPlus(3).upper(), "Three")

    def test_sum(self):                       <-- 2
        self.assertEqual((NumberPlus(2)+NumberPlus(5)).value, 7)

    def test_overflow(self):                  <-- 2
        with self.assertRaises(IndexError):
            NumberPlus(12)

if __name__ == "__main__":
    unittest.main()                           <-- 3
```

1. A “test case” is a class derived from `TestCase`
2. The test case is made up of a set of tests (methods named `test_...`)
3. Here’s where the test is run

Testing tools -1

Python provides a package to support the generation of testing procedures for python code:

unittest

file: `test_number.py` - A simple example

```
import unittest
from numberplus1 import NumberPlus

class TestNumber(unittest.TestCase):          <-- 1

    def test_upper(self):                     <-- 2
        self.assertTrue(NumberPlus(3).upper(), "Three")

    def test_sum(self):                       <-- 2
        self.assertEqual((NumberPlus(2)+NumberPlus(5)).value, 7)

    def test_overflow(self):                  <-- 2
        with self.assertRaises(IndexError):
            NumberPlus(12)

if __name__ == "__main__":
    unittest.main()                           <-- 3
```

1. A “test case” is a class derived from `TestCase`
2. The test case is made up of a set of tests (methods named `test_...`)
3. Here’s where the test is run

→

Testing tools -2

Now we run the test:

```
$ python test_number.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Testing tools -2

Now we run the test:

```
$ python test_number.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

The `unittest` package provides several other tools for building effective testing procedures:

Testing tools -2

Now we run the test:

```
$ python test_number.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

The `unittest` package provides several other tools for building effective testing procedures:

- Several `assertXxx()` methods

Testing tools -2

Now we run the test:

```
$ python test_number.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

The `unittest` package provides several other tools for building effective testing procedures:

- Several `assertXxx()` methods
- A `setUp()` method in `TestCase` will be executed before any test.

Testing tools -2

Now we run the test:

```
$ python test_number.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

The `unittest` package provides several other tools for building effective testing procedures:

- Several `assertXxx()` methods
- A `setUp()` method in `TestCase` will be executed before any test.
- A `tearDown()` method in `TestCase` will be executed after any test.

Testing tools -2

Now we run the test:

```
$ python test_number.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

The `unittest` package provides several other tools for building effective testing procedures:

- Several `assertXxx()` methods
- A `setUp()` method in `TestCase` will be executed before any test.
- A `tearDown()` method in `TestCase` will be executed after any test.
- A `TestSuite` class to group together related test cases.

Testing tools -2

Now we run the test:

```
$ python test_number.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

The `unittest` package provides several other tools for building effective testing procedures:

- Several `assertXxx()` methods
- A `setUp()` method in `TestCase` will be executed before any test.
- A `tearDown()` method in `TestCase` will be executed after any test.
- A `TestSuite` class to group together related test cases.
- A function `setUpModule()` will be called at the beginning of all tests

Testing tools -2

Now we run the test:

```
$ python test_number.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

The `unittest` package provides several other tools for building effective testing procedures:

- Several `assertXxx()` methods
- A `setUp()` method in `TestCase` will be executed before any test.
- A `tearDown()` method in `TestCase` will be executed after any test.
- A `TestSuite` class to group together related test cases.
- A function `setUpModule()` will be called at the beginning of all tests
- A function `tearDownModule()` will be called at the end of all tests

Testing tools -2

Now we run the test:

```
$ python test_number.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

The `unittest` package provides several other tools for building effective testing procedures:

- Several `assertXxx()` methods
- A `setUp()` method in `TestCase` will be executed before any test.
- A `tearDown()` method in `TestCase` will be executed after any test.
- A `TestSuite` class to group together related test cases.
- A function `setUpModule()` will be called at the beginning of all tests
- A function `tearDownModule()` will be called at the end of all tests

→

IDE for Python

An **I**ntegrated **D**evelopment **E**nvironment is an application integrating several of the programming tools quoted in previous slides. Here we quote only a few of them.

IDE for Python

An **I**ntegrated **D**evelopment **E**nvironment is an application integrating several of the programming tools quoted in previous slides. Here we quote only a few of them.

- **IDLE**

  The “standard” python IDE. It is itself written in Python so that it is available for any platform supporting Python [free]

IDE for Python

An **I**ntegrated **D**evelopment **E**nvironment is an application integrating several of the programming tools quoted in previous slides. Here we quote only a few of them.

- **IDLE**

  The “standard” python IDE. It is itself written in Python so that it is available for any platform supporting Python [free]
- **Eclipse**

  Eclipse is a multi language IDE supporting Python via the **PyDev** plugin. Available for Windows, Linux, MacOS [free].

IDE for Python

An **I**ntegrated **D**evelopment **E**nvironment is an application integrating several of the programming tools quoted in previous slides. Here we quote only a few of them.

- **IDLE**

  The “standard” python IDE. It is itself written in Python so that it is available for any platform supporting Python [free]
- **Eclipse**

  Eclipse is a multi language IDE supporting Python via the **PyDev** plugin. Available for Windows, Linux, MacOS [free].
- **Spyder**

  An IDE designed for scientific software development. Available for Windows, Linux, MacOS [free].

IDE for Python

An **I**ntegrated **D**evelopment **E**nvironment is an application integrating several of the programming tools quoted in previous slides. Here we quote only a few of them.

- **IDLE**
  The "standard" python IDE. It is itself written in Python so that it is available for any platform supporting Python [free]
- **Eclipse**
  Eclipse is a multi language IDE supporting Python via the **PyDev** plugin. Available for Windows, Linux, MacOS [free].
- **Spyder**
  An IDE designed for scientific software development. Available for Windows, Linux, MacOS [free].
- **Visual Studio**
  Has a Python plugin. Available for Windows only [proprietary]

IDE for Python

An **I**ntegrated **D**evelopment **E**nvironment is an application integrating several of the programming tools quoted in previous slides. Here we quote only a few of them.

- **IDLE**
  The "standard" python IDE. It is itself written in Python so that it is available for any platform supporting Python [free]
- **Eclipse**
  Eclipse is a multi language IDE supporting Python via the **PyDev** plugin. Available for Windows, Linux, MacOS [free].
- **Spyder**
  An IDE designed for scientific software development. Available for Windows, Linux, MacOS [free].
- **Visual Studio**
  Has a Python plugin. Available for Windows only [proprietary]

→

the Python Package Index

- PyPI is the main repository for Python packages and applications

pypi.python.org

the Python Package Index

- PyPI is the main repository for Python packages and applications

pypi.python.org

- The command to be used is **pip**[1]. E.g.:
  - **pip install astropy**
  - **pip list**
  - **pip uninstall astropy**

the Python Package Index

- PyPI is the main repository for Python packages and applications

pypi.python.org

- The command to be used is **pip**[1]. E.g.:
  - **pip install astropy**
  - **pip list**
  - **pip uninstall astropy**
- Sometimes the installation with `pip` will require the C/C++ compiler

the Python Package Index

- PyPI is the main repository for Python packages and applications

pypi.python.org

- The command to be used is **pip**[1]. E.g.:
  - **pip install astropy**
  - **pip list**
  - **pip uninstall astropy**
- Sometimes the installation with `pip` will require the C/C++ compiler
- Some suggestions:

the Python Package Index

- PyPI is the main repository for Python packages and applications

pypi.python.org

- The command to be used is **pip**[1]. E.g.:
  - **pip install astropy**
  - **pip list**
  - **pip uninstall astropy**
- Sometimes the installation with `pip` will require the C/C++ compiler
- Some suggestions:
  - If you find a standard package for your O.S. (e.g.: `msi` [Windows], `pkg` [MacOS], `rpm/deb` [Linux]) it may be better (i.e.: simpler) to use it.

the Python Package Index

- PyPI is the main repository for Python packages and applications

pypi.python.org

- The command to be used is **pip**[1]. E.g.:
  - **pip install astropy**
  - **pip list**
  - **pip uninstall astropy**
- Sometimes the installation with `pip` will require the C/C++ compiler
- Some suggestions:
  - If you find a standard package for your O.S. (e.g.: `msi` [Windows], `pkg` [MacOS], `rpm/deb` [Linux]) it may be better (i.e.: simpler) to use it.
  - Otherwise use **pip**.

the Python Package Index

- PyPI is the main repository for Python packages and applications

pypi.python.org

- The command to be used is **pip**[1]. E.g.:
  - **pip install astropy**
  - **pip list**
  - **pip uninstall astropy**
- Sometimes the installation with `pip` will require the C/C++ compiler
- Some suggestions:
  - If you find a standard package for your O.S. (e.g.: `msi` [Windows], `pkg` [MacOS], `rpm/deb` [Linux]) it may be better (i.e.: simpler) to use it.
  - Otherwise use **pip**.
  - **pip** is also recommended if you need the latest version of the package

the Python Package Index

- PyPI is the main repository for Python packages and applications

pypi.python.org

- The command to be used is **pip**[1]. E.g.:
  - **pip install astropy**
  - **pip list**
  - **pip uninstall astropy**
- Sometimes the installation with `pip` will require the C/C++ compiler
- Some suggestions:
  - If you find a standard package for your O.S. (e.g.: `msi` [Windows], `pkg` [MacOS], `rpm/deb` [Linux]) it may be better (i.e.: simpler) to use it.
  - Otherwise use **pip**.
  - **pip** is also recommended if you need the latest version of the package
  - Avoid to use both installation methods (maybe at different times)

the Python Package Index

- PyPI is the main repository for Python packages and applications

pypi.python.org

- The command to be used is **pip**[1]. E.g.:
  - **pip install astropy**
  - **pip list**
  - **pip uninstall astropy**
- Sometimes the installation with `pip` will require the C/C++ compiler
- Some suggestions:
  - If you find a standard package for your O.S. (e.g.: `msi` [Windows], `pkg` [MacOS], `rpm/deb` [Linux]) it may be better (i.e.: simpler) to use it.
  - Otherwise use **pip**.
  - **pip** is also recommended if you need the latest version of the package
  - Avoid to use both installation methods (maybe at different times)

[1] The command might sometimes be called **pip3**.

the Python Package Index

- PyPI is the main repository for Python packages and applications

pypi.python.org

- The command to be used is **pip**[1]. E.g.:
  - **pip install astropy**
  - **pip list**
  - **pip uninstall astropy**
- Sometimes the installation with `pip` will require the C/C++ compiler
- Some suggestions:
  - If you find a standard package for your O.S. (e.g.: `msi` [Windows], `pkg` [MacOS], `rpm/deb` [Linux]) it may be better (i.e.: simpler) to use it.
  - Otherwise use **pip**.
  - **pip** is also recommended if you need the latest version of the package
  - Avoid to use both installation methods (maybe at different times)

[1] The command might sometimes be called **pip3**.

→

IPython -1

IPython: a Python environment optimized for interactive use

It provides:

IPython -1

IPython: a Python environment optimized for interactive use

It provides:

- An enhanced version of the Python prompt
  - Full compatibility with Python
  - Powerful tab completion mechanism
  - History (persistent across sessions)
  - `%magic` commands

IPython -1

IPython: a Python environment optimized for interactive use

It provides:

- An enhanced version of the Python prompt
  - Full compatibility with Python
  - Powerful tab completion mechanism
  - History (persistent across sessions)
  - `%magic` commands
- Enhanced introspection tools
  - Standard Python's `help` system
  - Easy access to doc strings

IPython -1

IPython: a Python environment optimized for interactive use

It provides:

- An enhanced version of the Python prompt
  - Full compatibility with Python
  - Powerful tab completion mechanism
  - History (persistent across sessions)
  - `%magic` commands
- Enhanced introspection tools
  - Standard Python's `help` system
  - Easy access to doc strings
- Full integration with numerical and plotting packages
  - `numpy`, `scipy`
  - `matplotlib`

IPython -1

IPython: a Python environment optimized for interactive use

It provides:

- An enhanced version of the Python prompt
  - Full compatibility with Python
  - Powerful tab completion mechanism
  - History (persistent across sessions)
  - `%magic` commands
- Enhanced introspection tools
  - Standard Python's `help` system
  - Easy access to doc strings
- Full integration with numerical and plotting packages
  - `numpy`, `scipy`
  - `matplotlib`
- Support for parallel applications

IPython -1

IPython: a Python environment optimized for interactive use

It provides:

- An enhanced version of the Python prompt
  - Full compatibility with Python
  - Powerful tab completion mechanism
  - History (persistent across sessions)
  - `%magic` commands
- Enhanced introspection tools
  - Standard Python's `help` system
  - Easy access to doc strings
- Full integration with numerical and plotting packages
  - `numpy`, `scipy`
  - `matplotlib`
- Support for parallel applications
- At the core of the `Jupyter notebook`

IPython -1

IPython: a Python environment optimized for interactive use

It provides:

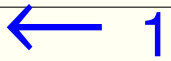
- An enhanced version of the Python prompt
  - Full compatibility with Python
  - Powerful tab completion mechanism
  - History (persistent across sessions)
  - `%magic` commands
- Enhanced introspection tools
  - Standard Python's `help` system
  - Easy access to doc strings
- Full integration with numerical and plotting packages
  - `numpy`, `scipy`
  - `matplotlib`
- Support for parallel applications
- At the core of the `Jupyter notebook`

→

```
$ ipython --pylab
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.3.0 -- An enhanced Interactive Python. Type '?' for help.
Using matplotlib backend: TkAgg

In [1]: a=3

In [2]: a+1
Out[2]: 4

In [3]:
```

IPython -2

```
$ ipython --pylab
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.3.0 -- An enhanced Interactive Python. Type '?' for help.
Using matplotlib backend: TkAgg

In [1]: a=3

In [2]: a+1
Out[2]: 4

In [3]:
```



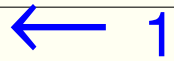
prompt

user input

ipython output

```
In [3]: %whos
Variable  Type   Data/Info
---------------------------
a         int    3

In [4]: !pwd
/home/lfini/Personale/CorsiSeminari/2019-Python
```

```
$ ipython --pylab
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.3.0 -- An enhanced Interactive Python. Type '?' for help.
Using matplotlib backend: TkAgg

In [1]: a=3

In [2]: a+1
Out[2]: 4

In [3]:
```

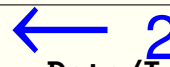


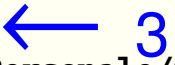
prompt

user input

ipython output

```
In [3]: %whos
Variable  Type   Data/Info
---------------------------
a         int    3

In [4]: !pwd
/home/lfini/Personale/CorsiSeminari/2019-Python
```





IPython -2

```
$ ipython --pylab   <-- 1
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.3.0 -- An enhanced Interactive Python. Type '?' for help.
Using matplotlib backend: TkAgg

In [1]: a=3

In [2]: a+1
Out[2]: 4

In [3]:
```

prompt

user input

ipython output

```
In [3]: %whos   <-- 2
Variable  Type   Data/Info
---------------------------
a         int    3

In [4]: !pwd   <-- 3
/home/lfini/Personale/CorsiSeminari/2019-Python
```

1. The **–pylab** option preloads `numpy` and `matplotlib` in ipython

IPython -2

```
$ ipython --pylab   <-- 1
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.3.0 -- An enhanced Interactive Python. Type '?' for help.
Using matplotlib backend: TkAgg

In [1]: a=3

In [2]: a+1
Out[2]: 4

In [3]:
```

prompt

user input

ipython output

```
In [3]: %whos   <-- 2
Variable  Type   Data/Info
---------------------------
a         int    3

In [4]: !pwd   <-- 3
/home/lfini/Personale/CorsiSeminari/2019-Python
```

1. The **–pylab** option preloads `numpy` and `matplotlib` in ipython
2. **%whos**: magic command - list currently defined variables

IPython -2

```
$ ipython --pylab   <-- 1
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.3.0 -- An enhanced Interactive Python. Type '?' for help.
Using matplotlib backend: TkAgg

In [1]: a=3

In [2]: a+1
Out[2]: 4

In [3]:
```

prompt

user input

ipython output

```
In [3]: %whos   <-- 2
Variable  Type   Data/Info
---------------------------
a         int    3

In [4]: !pwd   <-- 3
/home/lfini/Personale/CorsiSeminari/2019-Python
```

1. The **–pylab** option preloads `numpy` and `matplotlib` in ipython
2. **%whos**: magic command - list currently defined variables
3. **!pwd**: external shell command

IPython -2

```
$ ipython --pylab   <-- 1
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.3.0 -- An enhanced Interactive Python. Type '?' for help.
Using matplotlib backend: TkAgg

In [1]: a=3

In [2]: a+1
Out[2]: 4

In [3]:
```

prompt

user input

ipython output

```
In [3]: %whos   <-- 2
Variable  Type   Data/Info
---------------------------
a         int    3

In [4]: !pwd   <-- 3
/home/lfini/Personale/CorsiSeminari/2019-Python
```

1. The **–pylab** option preloads `numpy` and `matplotlib` in ipython
2. **%whos**: magic command - list currently defined variables
3. **!pwd**: external shell command

→

```
In [5]: ?np.max

Signature: np.max(a, axis=None, out=None, keepdims=<class 'numpy._globals._NoVal
Docstring:
Return the maximum of an array or maximum along an axis.

Parameters
----------
a : array_like
    Input data.
axis : None or int or tuple of ints, optional
    Axis or axes along which to operate. By default, flattened input is
    used.
    ...
In [6]: ??np.max
    ...
    ...
```

```
In [5]: ?np.max  ← 1,2

Signature: np.max(a, axis=None, out=None, keepdims=<class 'numpy._globals._NoVal
Docstring:
Return the maximum of an array or maximum along an axis.

Parameters
----------
a : array_like
    Input data.
axis : None or int or tuple of ints, optional
    Axis or axes along which to operate. By default, flattened input is
    used.
    ...
In [6]: ??np.max  ← 3
    ...
    ...
```

```
In [5]: ?np.max  ← 1,2

Signature: np.max(a, axis=None, out=None, keepdims=<class 'numpy._globals._NoVal
Docstring:
Return the maximum of an array or maximum along an axis.

Parameters
----------
a : array_like
    Input data.
axis : None or int or tuple of ints, optional
    Axis or axes along which to operate. By default, flattened input is
    used.
    ...
In [6]: ??np.max  ← 3
    ...
    ...
```

1. The package `numpy` is imported under the name **np**

IPython -2

```
In [5]: ?np.max  ← 1,2

Signature: np.max(a, axis=None, out=None, keepdims=<class 'numpy._globals._NoVal
Docstring:
Return the maximum of an array or maximum along an axis.

Parameters
----------
a : array_like
    Input data.
axis : None or int or tuple of ints, optional
    Axis or axes along which to operate. By default, flattened input is
    used.
    ...
In [6]: ??np.max  ← 3
    ...
    ...
```

1. The package `numpy` is imported under the name **np**
2. **?**: provides info about an object

IPython -2

```
In [5]: ?np.max  ← 1,2

Signature: np.max(a, axis=None, out=None, keepdims=<class 'numpy._globals._NoVal
Docstring:
Return the maximum of an array or maximum along an axis.

Parameters
----------
a : array_like
    Input data.
axis : None or int or tuple of ints, optional
    Axis or axes along which to operate. By default, flattened input is
    used.
    ...
In [6]: ??np.max  ← 3
    ...
    ...
```

1. The package `numpy` is imported under the name **np**
2. **?**: provides info about an object
3. **??**: provides more info (includes source code)

IPython -2

```
In [5]: ?np.max  ← 1,2

Signature: np.max(a, axis=None, out=None, keepdims=<class 'numpy._globals._NoVal
Docstring:
Return the maximum of an array or maximum along an axis.

Parameters
----------
a : array_like
    Input data.
axis : None or int or tuple of ints, optional
    Axis or axes along which to operate. By default, flattened input is
    used.
    ...
In [6]: ??np.max  ← 3
    ...
    ...
```

1. The package `numpy` is imported under the name **np**
2. **?**: provides info about an object
3. **??**: provides more info (includes source code)

→

Python scientific packages

- Scientific packages are usually not installed by default.

Python scientific packages

- Scientific packages are usually not installed by default.
- Installation procedures are several and may depend on the O.S. Here follow a few suggestions:

Python scientific packages

- Scientific packages are usually not installed by default.
- Installation procedures are several and may depend on the O.S. Here follow a few suggestions:
  - `ipython`, `numpy`, `scipy`, `matplotlib`
    - **Windows**: Can be installed from PyPI repository
    - **MacOS**: You may either use *Homebrew* or install from PyPI repository
    - **Linux**: All main distributions include scientific packages. If you want to have the latest version you may install from the PyPI repository.

Python scientific packages

- Scientific packages are usually not installed by default.
- Installation procedures are several and may depend on the O.S. Here follow a few suggestions:
  - `ipython`, `numpy`, `scipy`, `matplotlib`
    - **Windows**: Can be installed from PyPI repository
    - **MacOS**: You may either use *Homebrew* or install from PyPI repository
    - **Linux**: All main distributions include scientific packages. If you want to have the latest version you may install from the PyPI repository.
  - `astropy`:
    - **Windows**: Can be installed from PyPI repository
    - **MacOS**: You may either use *Homebrew* or install from PyPI repository
    - **Linux**: You may install from the distribution repository or from PyPI.

Python scientific packages

- Scientific packages are usually not installed by default.
- Installation procedures are several and may depend on the O.S. Here follow a few suggestions:
  - `ipython`, `numpy`, `scipy`, `matplotlib`
    - **Windows**: Can be installed from PyPI repository
    - **MacOS**: You may either use *Homebrew* or install from PyPI repository
    - **Linux**: All main distributions include scientific packages. If you want to have the latest version you may install from the PyPI repository.
  - `astropy`:
    - **Windows**: Can be installed from PyPI repository
    - **MacOS**: You may either use *Homebrew* or install from PyPI repository
    - **Linux**: You may install from the distribution repository or from PyPI.
  - `astroquery`:
    - **Windows**: Can be installed from PyPI repository
    - **MacOS**: You may either use *Homebrew* or install from PyPI repository
    - **Linux**: You may install from the distribution repository (but not all of them provide this package) or from PyPI.

Python scientific packages

- Scientific packages are usually not installed by default.
- Installation procedures are several and may depend on the O.S. Here follow a few suggestions:
  - `ipython`, `numpy`, `scipy`, `matplotlib`
    - **Windows**: Can be installed from PyPI repository
    - **MacOS**: You may either use *Homebrew* or install from PyPI repository
    - **Linux**: All main distributions include scientific packages. If you want to have the latest version you may install from the PyPI repository.
  - `astropy`:
    - **Windows**: Can be installed from PyPI repository
    - **MacOS**: You may either use *Homebrew* or install from PyPI repository
    - **Linux**: You may install from the distribution repository or from PyPI.
  - `astroquery`:
    - **Windows**: Can be installed from PyPI repository
    - **MacOS**: You may either use *Homebrew* or install from PyPI repository
    - **Linux**: You may install from the distribution repository (but not all of them provide this package) or from PyPI.

→

The End

End of Part III