Advanced Python for Astronomy

I'll present an overview of some advanced aspects of the Python programming language, with particular focus on scientific and astronomical packages.

- General aspects

Advanced Python for Astronomy

I’ll present an overview of some advanced aspects of the Python programming language, with particular focus on scientific and astronomical packages.

- General aspects
- Object Oriented Programming

Advanced Python for Astronomy

I’ll present an overview of some advanced aspects of the Python programming language, with particular focus on scientific and astronomical packages.

- General aspects
- Object Oriented Programming
- Linear algebra package: `numpy`

Advanced Python for Astronomy

I'll present an overview of some advanced aspects of the Python programming language, with particular focus on scientific and astronomical packages.

- General aspects
- Object Oriented Programming
- Linear algebra package: `numpy`
- Plotting package: `matplotlib`

Advanced Python for Astronomy

I’ll present an overview of some advanced aspects of the Python programming language, with particular focus on scientific and astronomical packages.

- General aspects
- Object Oriented Programming
- Linear algebra package: `numpy`
- Plotting package: `matplotlib`
- Scientific package: `scipy`

Advanced Python for Astronomy

I'll present an overview of some advanced aspects of the Python programming language, with particular focus on scientific and astronomical packages.

- General aspects
- Object Oriented Programming
- Linear algebra package: `numpy`
- Plotting package: `matplotlib`
- Scientific package: `scipy`
- Astronomical package: `astropy`

Advanced Python for Astronomy

I'll present an overview of some advanced aspects of the Python programming language, with particular focus on scientific and astronomical packages.

- General aspects
- Object Oriented Programming
- Linear algebra package: `numpy`
- Plotting package: `matplotlib`
- Scientific package: `scipy`
- Astronomical package: `astropy`
- Hands-on examples

Advanced Python for Astronomy

I’ll present an overview of some advanced aspects of the Python programming language, with particular focus on scientific and astronomical packages.

- General aspects
- Object Oriented Programming
- Linear algebra package: `numpy`
- Plotting package: `matplotlib`
- Scientific package: `scipy`
- Astronomical package: `astropy`
- Hands-on examples

→

Advanced Python for Astronomy

I'll present an overview of some advanced aspects of the Python programming language, with particular focus on scientific and astronomical packages.

- General aspects
- Object Oriented Programming
- Linear algebra package: `numpy`
- Plotting package: `matplotlib`
- Scientific package: `scipy`
- Astronomical package: `astropy`
- Hands-on examples

→

General aspects -1

- A scripting language

General aspects -1

- A scripting language
  - ... but not only
    - Numerical computations (!!)
    - CGI Applications
    - GUI programming
    - Gaming
    - Automation

General aspects -1

- A scripting language
  - ... but not only
    - Numerical computations (!!)
    - CGI Applications
    - GUI programming
    - Gaming
    - Automation
- Supports different programming models
  - Procedural
  - Object Oriented
  - Functional (... somewhat)

General aspects -1

- A scripting language
  - ... but not only
    - Numerical computations (!!)
    - CGI Applications
    - GUI programming
    - Gaming
    - Automation
- Supports different programming models
  - Procedural
  - Object Oriented
  - Functional (... somewhat)
- Semi-interpreted

General aspects -1

- A scripting language
  - ... but not only
    - Numerical computations (!!)
    - CGI Applications
    - GUI programming
    - Gaming
    - Automation
- Supports different programming models
  - Procedural
  - Object Oriented
  - Functional (... somewhat)
- Semi-interpreted
- Typed and dynamic
  - Typed objects
  - Non typed variables (names)
  - Automatic memory management

General aspects -1

- A scripting language
  - ... but not only
    - Numerical computations (!!)
    - CGI Applications
    - GUI programming
    - Gaming
    - Automation
- Supports different programming models
  - Procedural
  - Object Oriented
  - Functional (... somewhat)
- Semi-interpreted
- Typed and dynamic
  - Typed objects
  - Non typed variables (names)
  - Automatic memory management
- Huge standard library
  - Well portable

General aspects -1

- A scripting language
  - ... but not only
    - Numerical computations (!!)
    - CGI Applications
    - GUI programming
    - Gaming
    - Automation
- Supports different programming models
  - Procedural
  - Object Oriented
  - Functional (... somewhat)
- Semi-interpreted
- Typed and dynamic
  - Typed objects
  - Non typed variables (names)
  - Automatic memory management
- Huge standard library
  - Well portable
- Allows interactive use
  - Using the interpreter
  - Using `ipython`

General aspects -1

- A scripting language
  - ... but not only
    - Numerical computations (!!)
    - CGI Applications
    - GUI programming
    - Gaming
    - Automation
- Supports different programming models
  - Procedural
  - Object Oriented
  - Functional (... somewhat)
- Semi-interpreted
- Typed and dynamic
  - Typed objects
  - Non typed variables (names)
  - Automatic memory management
- Huge standard library
  - Well portable
- Allows interactive use
  - Using the interpreter
  - Using `ipython`

The competitors

Several other languages have similar characteristics or cover, at least partially, Python's areas of application.

- Java
- Perl
- Ruby
- Php
- R
- Matlab ($)
- Octave
- awk
- IDL ($)
- ....

→

The competitors

Several other languages have similar characteristics or cover, at least partially, Python's areas of application.

- Java
- Perl
- Ruby
- Php
- R
- Matlab ($)
- Octave
- awk
- IDL ($)
- ....

→

Versions and Platforms

- Versions
  - **3.7.0** - The “current” version[1]
  - **2.7.x** - Still widely used. Now frozen

Versions and Platforms

- Versions
  - **3.7.0** - The “current” version[1]
  - **2.7.x** - Still widely used. Now frozen
- Platforms
  - Linux (2.7.x preinstalled)
  - Windows and .NET
    - Installer at `www.python.org`
    - Active State (Commercial)
    - IronPython
  - Mac
    - Package from `www.python.org`
    - Can be installed with “homebrew”
  - Android
    - python-for-android + kivy[2]
    - buildozer[2]
    - SLA4

[1] As of July 2018

[2] For ”App” development

Versions and Platforms

- Versions
  - **3.7.0** - The “current” version[1]
  - **2.7.x** - Still widely used. Now frozen
- Platforms
  - Linux (2.7.x preinstalled)
  - Windows and .NET
    - Installer at `www.python.org`
    - Active State (Commercial)
    - IronPython
  - Mac
    - Package from `www.python.org`
    - Can be installed with “homebrew”
  - Android
    - python-for-android + kivy[2]
    - buildozer[2]
    - SLA4

[1] As of July 2018

[2] For ”App” development

In the following slides I’ll refer mainly to Python 3.x, adding some indications about differences with Python 2.x.

Versions and Platforms

- Versions
  - **3.7.0** - The “current” version[1]
  - **2.7.x** - Still widely used. Now frozen
- Platforms
  - Linux (2.7.x preinstalled)
  - Windows and .NET
    - Installer at `www.python.org`
    - Active State (Commercial)
    - IronPython
  - Mac
    - Package from `www.python.org`
    - Can be installed with “homebrew”
  - Android
    - python-for-android + kivy[2]
    - buildozer[2]
    - SLA4

---

[1] As of July 2018

[2] For ”App” development

> In the following slides I’ll refer mainly to Python 3.x, adding some indications about differences with Python 2.x.

General aspects -2

- Semi-interpreted language
  - Automatic p-code management
  - python2: `prog.py` → `prog.pyc` → `prog.pyo`
  - python3: ... in directory `__pycache__`

General aspects -2

- Semi-interpreted language
  - Automatic p-code management
  - python2: `prog.py` → `prog.pyc` → `prog.pyo`
  - python3: ... in directory `__pycache__`
- “Object Oriented” language
  - Full support of OO programming
  - “Everything is an object”

General aspects -2

- Semi-interpreted language
  - Automatic p-code management
  - python2: `prog.py` → `prog.pyc` → `prog.pyo`
  - python3: ... in directory `__pycache__`
- “Object Oriented” language
  - Full support of OO programming
  - “Everything is an object”
- “Typed” language
  - Objects have a type
  - Identifiers (names) are not typed

General aspects -2

- Semi-interpreted language
  - Automatic p-code management
  - python2: `prog.py` → `prog.pyc` → `prog.pyo`
  - python3: ... in directory `__pycache__`
- “Object Oriented” language
  - Full support of OO programming
  - “Everything is an object”
- “Typed” language
  - Objects have a type
  - Identifiers (names) are not typed
- Dynamic language
  - No declarations: object creation
  - Automatic “garbage collection”

General aspects -2

- Semi-interpreted language
  - Automatic p-code management
  - python2: `prog.py` → `prog.pyc` → `prog.pyo`
  - python3: ... in directory `__pycache__`
- “Object Oriented” language
  - Full support of OO programming
  - “Everything is an object”
- “Typed” language
  - Objects have a type
  - Identifiers (names) are not typed
- Dynamic language
  - No declarations: object creation
  - Automatic “garbage collection”

→

General aspects -2

- Semi-interpreted language
  - Automatic p-code management
  - python2: `prog.py` → `prog.pyc` → `prog.pyo`
  - python3: ... in directory `__pycache__`
- “Object Oriented” language
  - Full support of OO programming
  - “Everything is an object”
- “Typed” language
  - Objects have a type
  - Identifiers (names) are not typed
- Dynamic language
  - No declarations: object creation
  - Automatic “garbage collection”

→

General aspects -3

Dynamic object creation

```
>>> a=3


>>> a="three"




>>> p=[4,2,3,1]


>>> k=p
```

General aspects -3

Dynamic

Creates an object: “integer number valued 3” and gives it the name **a**

```
>>> a=3
```

```
>>> a="three"
```

```
>>> p=[4,2,3,1]
```

```
>>> k=p
```

General aspects -3

Dynamic

```
>>> a=3
```

Creates an object: “integer number

```
>>> a="three"
```

Creates an object “character string valued ”three”” and gives it the name **a**.
The name **a** is reused and the object “integer number ...” becomes unreachable

```
>>> p=[4,2,3,1]
```

```
>>> k=p
```

General aspects -3

Dynamic

```
>>> a=3
```

Creates an object: “integer number

```
>>> a="three"
```

Creates an object “character string valued ”three”” and gives it the name **a**.
The name **a** is reused and the object “integer number ...” becomes unreachable

```
>>> p=[4,2,3,1]
```

Creates an object “list of four elements” and gives it the name **p**

```
>>> k=p
```

General aspects -3

Dynamic

```
>>> a=3
```

Creates an object: “integer number

```
>>> a="three"
```

Creates an object “character string valued ”three”” and gives it the name **a**.
The name **a** is reused and the object “integer number ...” becomes unreachable

```
>>> p=[4,2,3,1]
```

Creates an object “list of four elements” and gives it the name **p**

```
>>> k=p
```

Gives another name (**k**) to the same object

General aspects -3

Dynamic

```
>>> a=3
```

Creates an object: “integer number

```
>>> a="three"
```

Creates an object “character string valued ”three”” and gives it the name **a**.
The name **a** is reused and the object “integer number ...” becomes unreachable

```
>>> p=[4,2,3,1]
```

Creates an object “list of four elements” and gives it the name **p**

```
>>> k=p
```

Gives another name (**k**) to the same object

Introspection

```
>>> type(a)
<type 'str'>
>>> type(p)
<type 'list'>

>>> dir(p)
...
>>> p.sort()
>>> k
[1, 2, 3, 4]
>>> del p[2]
>>> k
[1, 2, 4]
>>> help(k)
....
```

General aspects -3

Dynamic

```
>>> a=3
```

Creates an object: “integer number

```
>>> a="three"
```

Creates an object “character string valued ”three”” and gives it the name **a**.
The name **a** is reused and the object “integer number ...” becomes unreachable

```
>>> p=[4,2,3,1]
```

Creates an object “list of four elements” and gives it the name **p**

```
>>> k=p
```

Gives another name (**k**) to the same object

Introspection

```
>>> type(a)
<type 'str'>
>>> type(p)
<type 'list'>
```

The `type()` built-in function shows the object’s type

```
>>> dir(p)
...
>>> p.sort()
>>> k
[1, 2, 3, 4]
>>> del p[2]
>>> k
[1, 2, 4]
>>> help(k)
....
```

General aspects -3

Dynamic

```
>>> a=3
```

Creates an object: “integer number

```
>>> a="three"
```

Creates an object “character string valued ”three”” and gives it the name **a**.
The name **a** is reused and the object “integer number ...” becomes unreachable

```
>>> p=[4,2,3,1]
```

Creates an object “list of four elements” and gives it the name **p**

```
>>> k=p
```

Gives another name (**k**) to the same object

Introspection

```
>>> type(a)
<type 'str'>
>>> type(p)
<type 'list'>
```

The `type()` built-in function shows the object’s type

```
>>> dir(p)
...
>>> p.sort()
>>> k
[1, 2, 3, 4]
>>> del p[2]
>>> k
[1, 2, 4]
>>> help(k)
....
```

The `dir()` built-in function shows the object’s attributes

General aspects -3

Dynamic

```
>>> a=3
```

Creates an object: “integer number

```
>>> a="three"
```

Creates an object “character string valued ”three”” and gives it the name **a**.
The name **a** is reused and the object “integer number ...” becomes unreachable

```
>>> p=[4,2,3,1]
```

Creates an object “list of four elements” and gives it the name **p**

```
>>> k=p
```

Gives another name (**k**) to the same object

Introspection

```
>>> type(a)
<type 'str'>
>>> type(p)
<type 'list'>
```

The `type()` built-in function shows the object’s type

```
>>> dir(p)
...
```

The `dir()` built-in function shows the object’s attributes

```
>>> p.sort()
```

The `sort()` method sorts the list (in place)

```
>>> k
[1, 2, 3, 4]
>>> del p[2]
>>> k
[1, 2, 4]
>>> help(k)
....
```

General aspects -3

Dynamic

```
>>> a=3
```

Creates an object: “integer number

```
>>> a="three"
```

Creates an object “character string valued ”three”” and gives it the name **a**.
The name **a** is reused and the object “integer number ...” becomes unreachable

```
>>> p=[4,2,3,1]
```

Creates an object “list of four elements” and gives it the name **p**

```
>>> k=p
```

Gives another name (**k**) to the same object

Introspection

```
>>> type(a)
<type 'str'>
>>> type(p)
<type 'list'>
```

The `type()` built-in function shows the object’s type

```
>>> dir(p)
...
```

The `dir()` built-in function shows the object’s attributes

```
>>> p.sort()
```

The `sort()` method sorts the list (in place)

```
>>> k
[1, 2, 3, 4]
>>> del p[2]
```

The `del` statement deletes one element of the **p** list

```
>>> k
[1, 2, 4]
>>> help(k)
....
```

General aspects -3

Dynamic

```
>>> a=3
```

Creates an object: “integer number

```
>>> a="three"
```

Creates an object “character string valued ”three”” and gives it the name **a**.
The name **a** is reused and the object “integer number ...” becomes unreachable

```
>>> p=[4,2,3,1]
```

Creates an object “list of four elements” and gives it the name **p**

```
>>> k=p
```

Gives another name (**k**) to the same object

Introspection

```
>>> type(a)
<type 'str'>
>>> type(p)
<type 'list'>
```

The `type()` built-in function shows the object’s type

```
>>> dir(p)
...
```

The `dir()` built-in function shows the object’s attributes

```
>>> p.sort()
```

The `sort()` method sorts the list (in place)

```
>>> k
[1, 2, 3, 4]
>>> del p[2]
```

The `del` statement deletes one ele-

```
>>> k
[1, 2, 4]
```

This shows, again, that **k** and **p** are the same object

```
>>> help(k)
....
```

General aspects -3

Dynamic

```
>>> a=3
```

Creates an object: “integer number

```
>>> a="three"
```

Creates an object “character string valued ”three”” and gives it the name **a**.
The name **a** is reused and the object “integer number ...” becomes unreachable

```
>>> p=[4,2,3,1]
```

Creates an object “list of four elements” and gives it the name **p**

```
>>> k=p
```

Gives another name (**k**) to the same object

Introspection

```
>>> type(a)
<type 'str'>
>>> type(p)
<type 'list'>
```

The `type()` built-in function shows the object’s type

```
>>> dir(p)
...
```

The `dir()` built-in function shows the object’s attributes

```
>>> p.sort()
>>> k
[1, 2, 3, 4]
```

The `sort()` method sorts the list (in place)

```
>>> del p[2]
>>> k
[1, 2, 4]
```

The `del` statement deletes one ele-

```
>>> help(k)
....
```

The `help()` built-in function shows some object’s attributes in a “well printed” format

General aspects -3

Dynamic

```
>>> a=3
```

Creates an object: “integer number

```
>>> a="three"
```

Creates an object “character string valued ”three”” and gives it the name **a**.
The name **a** is reused and the object “integer number ...” becomes unreachable

```
>>> p=[4,2,3,1]
```

Creates an object “list of four elements” and gives it the name **p**

```
>>> k=p
```

Gives another name (**k**) to the same object

Introspection

```
>>> type(a)
<type 'str'>
>>> type(p)
<type 'list'>
```

The `type()` built-in function shows the object’s type

```
>>> dir(p)
...
```

The `dir()` built-in function shows the object’s attributes

```
>>> p.sort()
>>> k
[1, 2, 3, 4]
```

The `sort()` method sorts the list (in place)

```
>>> del p[2]
>>> k
[1, 2, 4]
```

The `del` statement deletes one ele-

```
>>> help(k)
....
```

The `help()` built-in function shows some object’s attributes in a “well printed” format

→

General aspects -4

Collections

```
>>> atuple = (1, 2, 4, "five", 6.0, -7, "VIII")
>>> alist = [1, "due", 3, "five", 7.96]
>>> adict = {1:"one", 2:2, 3:3.1415926, "five":5}
>>> aset = set(atuple)
```

General aspects -4

Collections

tuple: non mutable collection of objects, possibly different in type. Referenced by **index**.

```
>>> atuple = (1, 2, 4, "
>>> alist = [1, "due", 3, "five", 7.96]
>>> adict = {1:"one", 2:2, 3:3.1415926, "five":5}
>>> aset = set(atuple)
```

General aspects -4

Collections

```
>>> atuple = (1, 2, 4, "
>>> alist = [1, "due", 3,
>>> adict = {1:"one", 2:2,
>>> aset = set(atuple)
```

tuple: non mutable collection of ob-
je
e

list: mutable collection of objects, possibly different in type. Referenced by **index**

General aspects -4

Collections

```
>>> atuple = (1, 2, 4, "
>>> alist = [1, "due", 3,
>>> adict = {1:"one", 2:2,
>>> aset = set(atuple)
```

tuple: non mutable collection of ob-
je
e

list: mutable collection of objects,
p
b

dictionary: mutable collection of objects, possibly different in type. Referenced by **key**

General aspects -4

Collections

```
>>> atuple = (1, 2, 4, "
>>> alist = [1, "due", 3,
>>> adict = {1:"one", 2:2,
>>> aset = set(atuple)
```

tuple: non mutable collection of ob-
je
e

list: mutable collection of objects,
p
b

dictionary: mutable collection of objects, possibly different in type. Referenced by **key**

set: mutable collection of unique objects, (*frozenset*: non mutable). Supports usual operations on sets

General aspects -4

Collections

```
>>> atuple = (1, 2, 4, "
>>> alist = [1, "due", 3,
>>> adict = {1:"one", 2:2,
>>> aset = set(atuple)
```

tuple: non mutable collection of ob-

list: mutable collection of objects,

dictionary: mutable collection of objects, possibly different in type. Ref-

set: mutable collection of unique objects, (*frozenset*: non mutable). Supports usual operations on sets

From the Manual:

Function	Result
`x in s`	*True* if any element of **s** is equal to **x**, else *False*
`x not in s`	*False* if any element of **s** is equal to **x**, else *True*
`s+t`	concatenation of **s** and **t**
`s*n, n*s`	**s** + **s** + **s** + ... **n** times (**n** integer)
`s[i]`	i*th* element of **s** (base 0)
`s[i:j]`	slice of **s** from i*th* to (j-1)*th*
`s[i:j:k]`	slice of **s** from i*th* to (j-1)*th*, with stride k
`len(s)`	length (number of elements) of **s**
`min(s)`	the smallest element in **s**
`max(s)`	the greatest element in **s**
`s.index(x)`	index of the first occurrence of x in **s**
`s.count(x)`	number of occurrences of x in **s**

General aspects -4

Collections

```
>>> atuple = (1, 2, 4, "
>>> alist = [1, "due", 3,
>>> adict = {1:"one", 2:2,
>>> aset = set(atuple)
```

tuple: non mutable collection of ob-
je
e

list: mutable collection of objects,
p
b

dictionary: mutable collection of objects, possibly different in type. Referenced by **key**

set: mutable collection of unique objects, (*frozenset*: non mutable). Supports usual operations on sets

From the Manual:

Function	**Result**
`x in s` ⟸	*T* to **x**, else *False*
`x not in s`	*F* to **x**, else *True*
`s+t`	concatenation of **s** and **t**
`s*n, n*s`	**s** + **s** + **s** + ... **n** times (**n** integer)
`s[i]`	i*th* element of **s** (base 0)
`s[i:j]`	slice of **s** from i*th* to (j-1)*th*
`s[i:j:k]`	slice of **s** from i*th* to (j-1)*th*, with stride k
`len(s)`	length (number of elements) of **s**
`min(s)`	the smallest element in **s**
`max(s)`	the greatest element in **s**
`s.index(x)`	index of the first occurrence of x in **s**
`s.count(x)`	number of occurrences of x in **s**

What's the meaning if **s** is a dictionary?

General aspects -4

Collections

```
>>> atuple = (1, 2, 4, "
>>> alist = [1, "due", 3, "
>>> adict = {1:"one", 2:2, 
>>> aset = set(atuple)
```

tuple: non mutable collection of ob-
je
e

list: mutable collection of objects,
p
b

dictionary: mutable collection of objects, possibly different in type. Referenced by **key**

set: mutable collection of unique objects, (*frozenset*: non mutable). Supports usual operations on sets

From the Manual:

Function	Result
`x in s` ⟸	*T* o **x**, else *False*
`x not in s`	*F* to **x**, else *True*
`s+t`	concatenation of **s** and **t**
`s*n, n*s`	**s** + **s** + **s** + ... **n** times (**n** integer)
`s[i]`	i*th* element of **s** (base 0)
`s[i:j]`	slice of **s** from i*th* to (j-1)*th*
`s[i:j:k]`	slice of **s** from i*th* to (j-1)*th*, with stride k
`len(s)`	length (number of elements) of **s**
`min(s)`	the smallest element in **s**
`max(s)`	the greatest element in **s**
`s.index(x)`	index of the first occurrence of x in **s**
`s.count(x)`	number of occurrences of x in **s**

What's the meaning if **s** is a dictionary?

Comprehension

```
>>> another = [x*x for x in atuple if type(x) is int]
>>> atuple
(1, 2, 4, "cinque", 6.0, -7, "VIII")
>>> another
[1, 4, 16, 49]
```

General aspects -4

Collections

```
>>> atuple = (1, 2, 4, "
>>> alist = [1, "due", 3, "
>>> adict = {1:"one", 2:2, 
>>> aset = set(atuple)
```

tuple: non mutable collection of ob-
je
e

list: mutable collection of objects,
p
b

dictionary: mutable collection of objects, possibly different in type. Ref-
erenced by key

set: mutable collection of unique objects, (*frozenset*: non mutable). Supports usual operations on sets

From the Manual:

Function	Result
`x in s` ⟸	*T* o **x**, else *False*
`x not in s`	*F* to **x**, else *True*
`s+t`	concatenation of **s** and **t**
`s*n, n*s`	**s** + **s** + **s** + ... **n** times (**n** integer)
`s[i]`	i*th* element of **s** (base 0)
`s[i:j]`	slice of **s** from i*th* to (j-1)*th*
`s[i:j:k]`	slice of **s** from i*th* to (j-1)*th*, with stride k
`len(s)`	length (number of elements) of **s**
`min(s)`	the smallest element in **s**
`max(s)`	the greatest element in **s**
`s.index(x)`	index of the first occurrence of x in **s**
`s.count(x)`	number of occurrences of x in **s**

What's the meaning if **s** is a dictionary?

Comprehension

```
>>> another = [x*x for x in atuple if type(x) is int]
>>> atuple
(1, 2, 4, "cinque
>>> another
[1, 4, 16, 49]
```

Flexible syntax to create collections from other collections

General aspects -4

Collections

```
>>> atuple = (1, 2, 4, "
>>> alist = [1, "due", 3, "
>>> adict = {1:"one", 2:2, 
>>> aset = set(atuple)
```

tuple: non mutable collection of ob-
je
e

list: mutable collection of objects,
p
b

dictionary: mutable collection of objects, possibly different in type. Referenced by **key**

set: mutable collection of unique objects, (*frozenset*: non mutable). Supports usual operations on sets

From the Manual:

Function	Result
`x in s` ⟸	*T* o **x**, else *False*
`x not in s`	*F* to **x**, else *True*
`s+t`	concatenation of **s** and **t**
`s*n, n*s`	**s** + **s** + **s** + ... **n** times (**n** integer)
`s[i]`	i*th* element of **s** (base 0)
`s[i:j]`	slice of **s** from i*th* to (j-1)*th*
`s[i:j:k]`	slice of **s** from i*th* to (j-1)*th*, with stride k
`len(s)`	length (number of elements) of **s**
`min(s)`	the smallest element in **s**
`max(s)`	the greatest element in **s**
`s.index(x)`	index of the first occurrence of x in **s**
`s.count(x)`	number of occurrences of x in **s**

What's the meaning if **s** is a dictionary?

Comprehension

```
>>> another = [x*x for x in atuple if type(x) is int]
>>> atuple
(1, 2, 4, "cinque
>>> another
[1, 4, 16, 49]
```

Flexible syntax to create collections from other collections

Functions and Arguments

file: `funny.py`

```
def funny(a, b, default=3.1415926, toint=False):
    if toint:
        cvt = lambda x: int(x)
    else:
        cvt = lambda x: x
    if a > b:
        return cvt(a)
    elif a < b:
        return cvt(b)
    return cvt(default)
```

Functions and Arguments

Function **Definition**

file: `funny.py`

```
def funny(a, b, default=3.1415926, toint=False):
    if toint:
        cvt = lambda x: int(x)
    else:
        cvt = lambda x: x
    if a > b:
        return cvt(a)
    elif a < b:
        return cvt(b)
    return cvt(default)
```

Functions and Arguments

file: `funny.py`

Function **Definition**

```
def funny(a, b, default=3.1415926, toint=False):
    if toint:
        cvt = lambda x: int(x)
    else:
        cvt = lambda x: x
    if a > b:
        return cvt(a)
    elif a < b:
        return cvt(b)
    return cvt(default)
```

a, b: **positional** arguments

Functions and Arguments

file: `funny.py`

Function **Definition**

```
def funny(a, b, default=3.1415926, toint=False):
    if toint:
        cvt = lambda x: int(x)
    else:
        cvt = lambda x: x
    if a > b:
        return cvt(a)
    elif a < b:
        return cvt(b)
    return cvt(default)
```

a, b: **positional** arguments

`default`, `toint`: **named** (optional) arguments

`lambda`: “in place” function

Functions and Arguments

file: `funny.py`

Function **Definition**

```
def funny(a, b, default=3.1415926, toint=False):
    if toint:
        cvt = lambda x: int(x)
    else:
        cvt = lambda x: x
    if a > b:
        return cvt(a)
    elif a < b:
        return cvt(b)
    return cvt(default)
```

a, b: **positional** arguments

`default`, `toint`: **named** (optional) arguments

`lambda`: "in place" function

How to use functions:

```
>>> from funny import funny
>>> funny(1.0, 2.0)
2.0
>>> funny(1.0, 1.0)
3.1415926
>>> funny(1.0, 1.0, 6.2831852, toint=True)
6
>>> funny(1.0, 1.0, toint=True, default=6.2831852)
6
>>> funny(1.0, toint=True, default=6.283)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: funny() missing 1 required positional argument: ’b’
```

Functions and Arguments

file: `funny.py`

Function **Definition**

```
def funny(a, b, default=3.1415926, toint=False):
    if toint:
        cvt = lambda x: int(x)
    else:
        cvt = lambda x: x
    if a > b:
        return cvt(a)
    elif a < b:
        return cvt(b)
    return cvt(default)
```

a, b: **positional** arguments

`default`, `toint`: **named** (optional) arguments

`lambda`: “in place” function

How to use functions:

Function **call**

```
>>> from funny import funny
>>> funny(1.0, 2.0)
2.0
>>> funny(1.0, 1.0)
3.1415926
>>> funny(1.0, 1.0, 6.2831852, toint=True)
6
>>> funny(1.0, 1.0, toint=True, default=6.2831852)
6
>>> funny(1.0, toint=True, default=6.283)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: funny() missing 1 required positional argument: ’b’
```

Functions and Arguments

file: `funny.py`

Function **Definition**

```
def funny(a, b, default=3.1415926, toint=False):
    if toint:
        cvt = lambda x: int(x)
    else:
        cvt = lambda x: x
    if a > b:
        return cvt(a)
    elif a < b:
        return cvt(b)
    return cvt(default)
```

a, b: **positional** arguments

`default`, `toint`: **named** (optional) arguments

`lambda`: “in place” function

How to use functions:

Function **call**

```
>>> from funny import funny
>>> funny(1.0, 2.0)
2.0
>>> funny(1.0, 1.0)
3.1415926
>>> funny(1.0, 1.0, 6.2831852, toint=True)
6
>>> funny(1.0, 1.0, toint=True, default=6.2831852)
6
>>> funny(1.0, toint=True, default=6.283)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: funny() missing 1 required positional argument: ’b’
```

Only positional arguments

Functions and Arguments

file: `funny.py`

Function **Definition**

```
def funny(a, b, default=3.1415926, toint=False):
    if toint:
        cvt = lambda x: int(x)
    else:
        cvt = lambda x: x
    if a > b:
        return cvt(a)
    elif a < b:
        return cvt(b)
    return cvt(default)
```

a, b: **positional** arguments

`default`, `toint`: **named** (optional) arguments

`lambda`: “in place” function

How to use functions:

Function **call**

```
>>> from funny import funny
>>> funny(1.0, 2.0)
2.0
>>> funny(1.0, 1.0)
3.1415926
>>> funny(1.0, 1.0, 6.2831852, toint=True)
6
>>> funny(1.0, 1.0, toint=True, default=6.2831852)
6
>>> funny(1.0, toint=True, default=6.283)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: funny() missing 1 required positional argument: 'b'
```

Only positional a

positional arguments, one optional argument (specified by position), one optional argument (specified by name)

Functions and Arguments

file: `funny.py`

Function **Definition**

```
def funny(a, b, default=3.1415926, toint=False):
    if toint:
        cvt = lambda x: int(x)
    else:
        cvt = lambda x: x
    if a > b:
        return cvt(a)
    elif a < b:
        return cvt(b)
    return cvt(default)
```

a, b: **positional** arguments

`default`, `toint`: **named** (optional) arguments

`lambda`: “in place” function

How to use functions:

Function **call**

```
>>> from funny import funny
>>> funny(1.0, 2.0)
2.0
>>> funny(1.0, 1.0)
3.1415926
>>> funny(1.0, 1.0, 6.2831852, toint=True)
6
>>> funny(1.0, 1.0, toint=True, default=6.28318
6
>>> funny(1.0, toint=True, default=6.283)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: funny() missing 1 required positional argument: 'b'
```

Only positional a

positional arguments, one optional argument (specified by position), one optional argument (specified by name)

Positional arguments, named arguments (order irrelevant)

Functions and Arguments

file: `funny.py`

Function **Definition**

```
def funny(a, b, default=3.1415926, toint=False):
    if toint:
        cvt = lambda x: int(x)
    else:
        cvt = lambda x: x
    if a > b:
        return cvt(a)
    elif a < b:
        return cvt(b)
    return cvt(default)
```

a, b: **positional** arguments

`default`, `toint`: **named** (optional) arguments

`lambda`: “in place” function

How to use functions:

Function **call**

```
>>> from funny import funny
>>> funny(1.0, 2.0)
2.0
>>> funny(1.0, 1.0)
3.1415926
>>> funny(1.0, 1.0, 6.2831852, toint=True)
6
>>> funny(1.0, 1.0, toint=True, default=6.28318
6
>>> funny(1.0, toint=True, default=6.283)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: funny() missing 1 required positional argument: 'b'
```

⇐ Only positional a

⇐ positional arguments, one optional argument (specified by position), one optional argument (specified by name)

⇐ Positional arguments, named arguments (order irrelevant)

⇐ Positional arguments are required

Functions and Arguments

file: `funny.py`

Function **Definition**

```
def funny(a, b, default=3.1415926, toint=False):
    if toint:
        cvt = lambda x: int(x)
    else:
        cvt = lambda x: x
    if a > b:
        return cvt(a)
    elif a < b:
        return cvt(b)
    return cvt(default)
```

a, b: **positional** arguments

`default`, `toint`: **named** (optional) arguments

`lambda`: “in place” function

How to use functions:

Function **call**

```
>>> from funny import funny
>>> funny(1.0, 2.0)
2.0
>>> funny(1.0, 1.0)
3.1415926
>>> funny(1.0, 1.0, 6.2831852, toint=True)
6
>>> funny(1.0, 1.0, toint=True, default=6.28318
6
>>> funny(1.0, toint=True, default=6.283)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: funny() missing 1 required positional argument: 'b'
```

Only positional a

positional arguments, one optional argument (specified by position), one optional argument (specified by name)

Positional arguments, named arguments (order irrelevant)

Positional arguments are required

In summary:

Functions and Arguments

file: `funny.py`

Function **Definition**

```
def funny(a, b, default=3.1415926, toint=False):
    if toint:
        cvt = lambda x: int(x)
    else:
        cvt = lambda x: x
    if a > b:
        return cvt(a)
    elif a < b:
        return cvt(b)
    return cvt(default)
```

a, b: **positional** arguments

`default`, `toint`: **named** (optional) arguments

`lambda`: “in place” function

How to use functions:

Function **call**

```
>>> from funny import funny
>>> funny(1.0, 2.0)
2.0
>>> funny(1.0, 1.0)
3.1415926
>>> funny(1.0, 1.0, 6.2831852, toint=True)
6
>>> funny(1.0, 1.0, toint=True, default=6.28318
6
>>> funny(1.0, toint=True, default=6.283)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: funny() missing 1 required positional argument: 'b'
```

Only positional a

positional arguments, one optional argument (specified by position), one optional argument (specified by name)

Positional arguments, named arguments (order irrelevant)

Positional arguments are required

In summary:

- Positional arguments precede named ones

Functions and Arguments

file: `funny.py`

Function **Definition**

```
def funny(a, b, default=3.1415926, toint=False):
    if toint:
        cvt = lambda x: int(x)
    else:
        cvt = lambda x: x
    if a > b:
        return cvt(a)
    elif a < b:
        return cvt(b)
    return cvt(default)
```

a, b: **positional** arguments

`default`, `toint`: **named** (optional) arguments

`lambda`: “in place” function

How to use functions:

Function **call**

```
>>> from funny import funny
>>> funny(1.0, 2.0)
2.0
>>> funny(1.0, 1.0)
3.1415926
>>> funny(1.0, 1.0, 6.2831852, toint=True)
6
>>> funny(1.0, 1.0, toint=True, default=6.28318
6
>>> funny(1.0, toint=True, default=6.283)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: funny() missing 1 required positional argument: 'b'
```

Only positional a

positional arguments, one optional argument (specified by position), one optional argument (specified by name)

Positional arguments, named arguments (order irrelevant)

Positional arguments are required

In summary:

- Positional arguments precede named ones
- Positional arguments are required

Functions and Arguments

file: funny.py

Function **Definition**

```python
def funny(a, b, default=3.1415926, toint=False):
    if toint:
        cvt = lambda x: int(x)
    else:
        cvt = lambda x: x
    if a > b:
        return cvt(a)
    elif a < b:
        return cvt(b)
    return cvt(default)
```

a, b: **positional** arguments

`default`, `toint`: **named** (optional) arguments

`lambda`: “in place” function

How to use functions:

Function **call**

```
>>> from funny import funny
>>> funny(1.0, 2.0)
2.0
>>> funny(1.0, 1.0)
3.1415926
>>> funny(1.0, 1.0, 6.2831852, toint=True)
6
>>> funny(1.0, 1.0, toint=True, default=6.28318
6
>>> funny(1.0, toint=True, default=6.283)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: funny() missing 1 required positional argument: ’b’
```

Only positional a

positional arguments, one optional argument (specified by position), one optional argument (specified by name)

Positional arguments, named arguments (order irrelevant)

Positional arguments are required

In summary:

- Positional arguments precede named ones
- Positional arguments are required
- Named arguments are optional and have a default value

Functions and Arguments

file: `funny.py`

Function **Definition**

```
def funny(a, b, default=3.1415926, toint=False):
    if toint:
        cvt = lambda x: int(x)
    else:
        cvt = lambda x: x
    if a > b:
        return cvt(a)
    elif a < b:
        return cvt(b)
    return cvt(default)
```

a, b: **positional** arguments

`default`, `toint`: **named** (optional) arguments

`lambda`: “in place” function

How to use functions:

Function **call**

```
>>> from funny import funny
>>> funny(1.0, 2.0)
2.0
>>> funny(1.0, 1.0)
3.1415926
>>> funny(1.0, 1.0, 6.2831852, toint=True)
6
>>> funny(1.0, 1.0, toint=True, default=6.28318
6
>>> funny(1.0, toint=True, default=6.283)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: funny() missing 1 required positional argument: ’b’
```

Only positional a

positional arguments, one optional argument (specified by position), one optional argument (specified by name)

Positional arguments, named arguments (order irrelevant)

Positional arguments are required

In summary:

- Positional arguments precede named ones
- Positional arguments are required
- Named arguments are optional and have a default value
- Named arguments may be specified either by position or by name.

Functions and Arguments

file: `funny.py`

Function **Definition**

```
def funny(a, b, default=3.1415926, toint=False):
    if toint:
        cvt = lambda x: int(x)
    else:
        cvt = lambda x: x
    if a > b:
        return cvt(a)
    elif a < b:
        return cvt(b)
    return cvt(default)
```

a, b: **positional** arguments

`default`, `toint`: **named** (optional) arguments

`lambda`: “in place” function

How to use functions:

Function **call**

```
>>> from funny import funny
>>> funny(1.0, 2.0)
2.0
>>> funny(1.0, 1.0)
3.1415926
>>> funny(1.0, 1.0, 6.2831852, toint=True)
6
>>> funny(1.0, 1.0, toint=True, default=6.28318
6
>>> funny(1.0, toint=True, default=6.283)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: funny() missing 1 required positional argument: 'b'
```

Only positional a

positional arguments, one optional argument (specified by position), one optional argument (specified by name)

Positional arguments, named arguments (order irrelevant)

Positional arguments are required

In summary:

- Positional arguments precede named ones
- Positional arguments are required
- Named arguments are optional and have a default value
- Named arguments may be specified either by position or by name.
- When specified by name, order is irrelevant

Variable arguments -1

Python provides syntax to write functions which can be called with a variable number of arguments, both positional and named.

Let's see the general case:

file: `showarg.py`

```
def showarg(a, b, d=15, *pos, **kw):
    print("Positional required arguments (a,b):", a, b)
    print("Standard named arguments (d):", d)
    print("Positional variable arguments (pos):", pos)
    print("Named variable arguments (kw):", kw)
```

Variable arguments -1

Python provides syntax to write functions which can be called with a variable number of arguments, both positional and named.

Let's see the general case:

file: `showarg.py`

```
def showarg(a, b, d=15, *pos, **kw):
    print("Positional required arguments (a,b):", a, b)
    print("Standard named arguments (d):", d)
    print("Positional variable arguments (pos):", pos)
    print("Named variable arguments (kw):", kw)
```

Variable arguments -1

Python provides syntax to write functions which can be called with a variable number of arguments, both positional and named.

Let's see the general case:

file: `showarg.py`

```
def showarg(a, b, d=15, *pos, **kw):
    print("Positional required
    print("Standard named argu
    print("Positional variable
    print("Named variable argu
```

The **showarg()** function only shows the arguments from the call.

Variable arguments -1

Python provides syntax to write functions which can be called with a variable number of arguments, both positional and named.

Let's see the general case:

file: `showarg.py`

```
def showarg(a, b, d=15, *pos, **kw):
    print("Positional required
    print("Standard named argu
    print("Positional variable
    print("Named variable argu
```

The **showarg()** function only

a, **b** standard positional arguments

Variable arguments -1

Python provides syntax to write functions which can be called with a variable number of arguments, both positional and named.

Let's see the general case:

file: `showarg.py`

```
def showarg(a, b, d=15, *pos, **kw):
    print("Positional required
    print("Standard named argu
    print("Positional variable
    print("Named variable argu
```

The **showarg()** function only

a, **b** standard positional argu-

d, standard named argument

Variable arguments -1

Python provides syntax to write functions which can be called with a variable number of arguments, both positional and named.

Let's see the general case:

file: `showarg.py`

```
def showarg(a, b, d=15, *pos, **kw):
    print("Positional required
    print("Standard named argu
    print("Positional variable
    print("Named variable argu
```

The **showarg()** function only

a, **b** standard positional argu

***pos**, (*tuple*) variable positional arguments

Variable arguments -1

Python provides syntax to write functions which can be called with a variable number of arguments, both positional and named.

Let's see the general case:

file: `showarg.py`

```
def showarg(a, b, d=15, *pos, **kw):
    print("Positional required
    print("Standard named argu
    print("Positional variable
    print("Named variable argu
```

The **showarg()** function only

a, **b** standard positional argu-

****kw**, (*dict*) variable named arguments

Variable arguments -1

Python provides syntax to write functions which can be called with a variable number of arguments, both positional and named.

Let's see the general case:

file: `showarg.py`

```
def showarg(a, b, d=15, *pos, **kw):
    print("Positional required
    print("Standard named argu
    print("Positional variable
    print("Named variable argu
```

The **showarg()** function only

a, **b** standard positional argu

****kw**, (*dict*) variable named arguments

Calling `showarg()`:

```
>>> from showarg import showarg

>>> showarg(1, 2)
Positional required arguments (a,b): 1 2
Standard named arguments (d): 15
Positional variable arguments (pos): ()
Named variable arguments (kw): {}

>>> showarg(1, 2, 3, 4, 5, 6, opt1=7, opt2=8)
Positional required arguments (a,b): 1 2
Standard named arguments (d): 3
Positional variable arguments (pos): (4, 5, 6)
Named variable arguments (kw): {'opt1': 7, 'opt2': 8}
```

Variable arguments -1

Python provides syntax to write functions which can be called with a variable number of arguments, both positional and named.

Let's see the general case:

file: `showarg.py`

```
def showarg(a, b, d=15, *pos, **kw):
    print("Positional required
    print("Standard named argu
    print("Positional variable
    print("Named variable argu
```

The **showarg()** function only

a, **b** standard positional argu

****kw**, (*dict*) variable named arguments

Calling `showarg()`:

```
>>> from showarg import showarg

>>> showarg(1, 2)
Positional required arguments (a,b): 1 2
Standard named arguments (d): 15
Positional variable arguments (pos): ()
Named variable arguments (kw): {}

>>> showarg(1, 2, 3, 4, 5, 6, opt1=7, opt2=8)
Positional required arguments (a,b): 1 2
Standard named arguments (d): 3
Positional variable arguments (pos): (4, 5, 6)
Named variable arguments (kw): {'opt1': 7, 'opt2': 8}
```

Call with required arguments only

Variable arguments -1

Python provides syntax to write functions which can be called with a variable number of arguments, both positional and named.

Let's see the general case:

file: showarg.py

```
def showarg(a, b, d=15, *pos, **kw):
    print("Positional required
    print("Standard named argu
    print("Positional variable
    print("Named variable argu
```

The **showarg()** function only

a **b** standard positional argu

****kw**, (*dict*) variable named arguments

Calling showarg():

```
>>> from showarg import showarg

>>> showarg(1, 2)
Positional required arguments (a,b): 1 2
Standard named arguments (d): 15
Positional variable arguments (pos): ()
Named variable arguments (kw): {}
```

Call with required arguments only

```
>>> showarg(1, 2, 3, 4, 5, 6, opt1=7, opt2=8)
Positional required arguments (a,b): 1 2
Standard named arguments (d): 3
Positional variable arguments (pos): (4, 5, 6)
Named variable arguments (kw): {'opt1': 7, 'opt2': 8}
```

Call with variable positional and named arguments

Variable arguments -1

Python provides syntax to write functions which can be called with a variable number of arguments, both positional and named.

Let’s see the general case:

file: `showarg.py`

```
def showarg(a, b, d=15, *pos, **kw):
    print("Positional required
    print("Standard named argu
    print("Positional variable
    print("Named variable argu
```

The **showarg()** function only

a, **b** standard positional argu

****kw**, (*dict*) variable named arguments

Calling `showarg()`:

```
>>> from showarg import showarg

>>> showarg(1, 2)
Positional required arguments (a,b): 1 2
Standard named arguments (d): 15
Positional variable arguments (pos): ()
Named variable arguments (kw): {}

>>> showarg(1, 2, 3, 4, 5, 6, opt1=7, opt2=8)
Positional required arguments (a,b): 1 2
Standard named arguments (d): 3
Positional variable arguments (pos): (4, 5, 6)
Named variable arguments (kw): {'opt1': 7, 'opt2': 8}
```

⇐ Call with required arguments only

⇐ Call with variable positional and named arguments

⇐ Note!

Variable arguments -1

Python provides syntax to write functions which can be called with a variable number of arguments, both positional and named.

Let's see the general case:

file: `showarg.py`

```
def showarg(a, b, d=15, *pos, **kw):
    print("Positional required
    print("Standard named argu
    print("Positional variable
    print("Named variable argu
```

The **showarg()** function only

a **b** standard positional argu

****kw**, (*dict*) variable named arguments

Calling `showarg()`:

```
>>> from showarg import showarg

>>> showarg(1, 2)
Positional required arguments (a,b): 1 2
Standard named arguments (d): 15
Positional variable arguments (pos): ()
Named variable arguments (kw): {}

>>> showarg(1, 2, 3, 4, 5, 6, opt1=7, opt2=8)
Positional required arguments (a,b): 1 2
Standard named arguments (d): 3
Positional variable arguments (pos): (4, 5, 6)
Named variable arguments (kw): {'opt1': 7, 'opt2': 8}
```

⇐ Call with required arguments only

⇐ Call with variable positional and named arguments

⇐ Note!

→

Variable arguments -2

Python allows to put arbitrary argument lists in the function call with a syntax analogous to variable argument lists in function definition.

Another way to call `showarg()`:

```
>>> ps=(10,11,12)
>>> ak={"arg1":13, "arg2":14, "arg3":15}

>>> showarg(1, 2, 3, 4, 5, 6, opt1=7, opt2=8, *ps, **ak)
Positional required arguments (a,b): 1 2
Standard named arguments (d): 3
Positional variable arguments (pos): (4, 5, 6, 10, 11, 12)
Named variable arguments (kw): {'opt1': 7, 'opt2': 8, 'arg3': 15,
 'arg2': 14, 'arg1': 13}
```

Variable arguments -2

Python allows to put arbitrary argument lists in the function call with a syntax analogous to variable argument lists in function definition.

Another way to call `showarg()`:

```
>>> ps=(10,11,12)
>>> ak={"arg1":13, "arg2":14, "arg3":15}

>>> showarg(1, 2, 3, 4, 5, 6, opt1=7, opt2=8, *ps, **ak)
Positional required arguments (a,b): 1 2
Standard named arguments (d): 3
Positional variable arguments (pos): (4, 5, 6, 10, 11, 12)
Named variable arguments (kw): {'opt1': 7, 'opt2': 8, 'arg3': 15,
 'arg2': 14, 'arg1': 13}
```

Variable arguments -2

Python allows to put arbitrary argument lists in the function call with a syntax analogous to variable argument lists in function definition.

Another way to call `showarg()`:

Definition of a *tuple* (**ps**) and a *dictionary* (**ak**)

```
>>> ps=(10,11,12)
>>> ak={"arg1":13, "arg2":14, "arg3":15}

>>> showarg(1, 2, 3, 4, 5, 6, opt1=7, opt2=8, *ps, **ak)
Positional required arguments (a,b): 1 2
Standard named arguments (d): 3
Positional variable arguments (pos): (4, 5, 6, 10, 11, 12)
Named variable arguments (kw): {'opt1': 7, 'opt2': 8, 'arg3': 15,
 'arg2': 14, 'arg1': 13}
```

Variable arguments -2

Python allows to put arbitrary argument lists in the function call with a syntax analogous to variable argument lists in function definition.

Another way to call `showarg()`:

```
>>> ps=(10,11,12)
>>> ak={"arg1":13, "arg2":14, "arg3":15}

>>> showarg(1, 2, 3, 4, 5, 6, opt1=7, opt2=8, *ps, **ak)
Positional required arguments (a,b): 1 2
Standard named arguments (d): 3
Positional variable arguments (pos): (4,
Named variable arguments (kw): {’opt1’: 7
 ’arg2’: 14, ’arg1’: 13}
```

Definition of a *tuple* (**ps**) and a *dictionary* (**ak**)

The **pos** *tuple* in the function “receives” positional parameters other than required ones and the content of **ps** *tuple* from the call

Variable arguments -2

Python allows to put arbitrary argument lists in the function call with a syntax analogous to variable argument lists in function definition.

Another way to call `showarg()`:

```
>>> ps=(10,11,12)
>>> ak={"arg1":13, "arg2":14, "arg3":15}

>>> showarg(1, 2, 3, 4, 5, 6, opt1=7, opt2=8, *ps, **ak)
Positional required arguments (a,b): 1 2
Standard named arguments (d): 3
Positional variable arguments (pos): (4,
Named variable arguments (kw): {'opt1': 7
 'arg2': 14, 'arg1': 13}
```

Definition of a *tuple* (**ps**) and a *dictionary* (**ak**)

The **pos** *tuple* in the function “receives” posi-
ti
t
t
f

The **kw** *dictionary* in the function “receives” named variable arguments and the content of **ak** *dictionary* from the call

Variable arguments -2

Python allows to put arbitrary argument lists in the function call with a syntax analogous to variable argument lists in function definition.

Another way to call `showarg()`:

```
>>> ps=(10,11,12)
>>> ak={"arg1":13, "arg2":14, "arg3":15}

>>> showarg(1, 2, 3, 4, 5, 6, opt1=7, opt2=8, *ps, **ak)
Positional required arguments (a,b): 1 2
Standard named arguments (d): 3
Positional variable arguments (pos): (4,
Named variable arguments (kw): {'opt1': 7
 'arg2': 14, 'arg1': 13}
```

Definition of a *tuple* (**ps**) and a *dictionary* (**ak**)

The **pos** *tuple* in the function “receives” posi-
ti
t
t
f

The **kw** *dictionary* in the function “receives” named variable arguments and the content of **ak** *dictionary* from the call

→

Modules -1

Module: A block of code contained in a single file

file: `fibo.py`

```
"Module for the computation of Fibonacci series"

MAXFIBO=1000

def fibo(n):
    "Returns the Fibonacci series up to n"
    if n > MAXFIBO: return []
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result

def fibo_print(n):
    "prints the Fibonacci series up to n"
    ser = fibo(n)
    print("Fibonacci series up to %d:" % n)
    print(ser)

if __name__=='__main__':
    fibo_print(14)
```

Modules -1

Module: A block of code contained in a single file

file: `fibo.py`

```
"Module for the computation of Fibonacci series"

MAXFIBO=1000

def fibo(n):
    "Returns the Fibonacci series up to n"
    if n > MAXFIBO: return []
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result

def fibo_print(n):
    "prints the Fibonacci series up to n"
    ser = fibo(n)
    print("Fibonacci series up to %d:" % n)
    print(ser)

if __name__=='__main__':
    fibo_print(14)
```

Modules -1

Module: A block of code contained in a single file

file: `fibo.py`

```
"Module for the computation of Fibonacci series"

MAXFIBO=1000

def fibo(n):
    "Returns the Fibonacci series up to n"
    if n > MAXFIBO: return []
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result

def fibo_print(n):
    "prints the Fibonacci series up to n"
    ser = fibo(n)
    print("Fibonacci series up to %d:" % n)
    print(ser)

if __name__=='__main__':
    fibo_print(14)
```

⇐ The constant **MAXFIBO**

Modules -1

Module: A block of code contained in a single file

file: `fibo.py`

```
"Module for the computation of Fibonacci series"

MAXFIBO=1000

def fibo(n):
    "Returns the Fibonacci series up to n"
    if n > MAXFIBO: return []
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result

def fibo_print(n):
    "prints the Fibonacci series up to n"
    ser = fibo(n)
    print("Fibonacci series up to %d:" % n)
    print(ser)

if __name__=='__main__':
    fibo_print(14)
```

The constant **MAXFIBO**

The function **fibo()**

Modules -1

Module: A block of code contained in a single file

file: `fibo.py`

```
"Module for the computation of Fibonacci series"

MAXFIBO=1000

def fibo(n):
    "Returns the Fibonacci series up to n"
    if n > MAXFIBO: return []
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result

def fibo_print(n):
    "prints the Fibonacci series up to n"
    ser = fibo(n)
    print("Fibonacci series up to %d:" % n)
    print(ser)

if __name__=='__main__':
    fibo_print(14)
```

⇐ The constant **MAXFIBO**

⇐ The function **fibo()**

⇐ The function **fibo_print()**

Modules -1

Module: A block of code contained in a single file

file: `fibo.py`

```
"Module for the computation of Fibonacci series"

MAXFIBO=1000

def fibo(n):
    "Returns the Fibonacci series up to n"
    if n > MAXFIBO: return []
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result

def fibo_print(n):
    "prints the Fibonacci series up to n"
    ser = fibo(n)
    print("Fibonacci series up to %d:" % n)
    print(ser)

if __name__=='__main__':
    fibo_print(14)
```

The constant **MAXFIBO**

The function **fibo()**

The function **fibo_print()**

Some test code

Modules -1

Module: A block of code contained in a single file

file: `fibo.py`

```python
"Module for the computation of Fibonacci series"

MAXFIBO=1000

def fibo(n):
    "Returns the Fibonacci series up to n"
    if n > MAXFIBO: return []
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result

def fibo_print(n):
    "prints the Fibonacci series up to n"
    ser = fibo(n)
    print("Fibonacci series up to %d:" % n)
    print(ser)

if __name__=='__main__':
    fibo_print(14)
```

The constant **MAXFIBO**

The function **fibo()**

The function **fibo_print()**

Some test code

Note: The internal variable `__name__` holds the string `"__main__"` only when the code is directly executed. If, instead, the module is *imported*, as we will see in the next slide, the variable holds the name of the module (`fibo`).
As a consequence when the module is imported, the test code is not executed.

Modules -2

```
>>> import fibo

>>> fibo.fibo(400)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]

>>> dir(fibo)
['MAXFIBO', '__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__', 'fibo', 'fibo_print']

>>> help(fibo)
Help on module fibo:

NAME
    fibo - Module for the computation of Fibonacci series

FUNCTIONS
    fibo(n)
        Returns the Fibonacci series up to n

    fibo_print(n)
        prints the Fibonacci series up to n

DATA
    MAXFIBO = 1000

FILE
    /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py

>>> fibo.__name__
'fibo'

>>> fibo.__doc__
'Module for the computation of Fibonacci series'

>>> fibo.__file__
'/home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py'

>>> fibo.MAXFIBO
1000

>>> fibo.fibo_print(123)
Fibonacci series up to 123:
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Modules -2

To use a module it must be **imported**

```
>>> import fibo

>>> fibo.fibo(400)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]

>>> dir(fibo)
['MAXFIBO', '__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__', 'fibo', 'fibo_print']

>>> help(fibo)
Help on module fibo:

NAME
    fibo - Module for the computation of Fibonacci series

FUNCTIONS
    fibo(n)
        Returns the Fibonacci series up to n

    fibo_print(n)
        prints the Fibonacci series up to n

DATA
    MAXFIBO = 1000

FILE
    /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py

>>> fibo.__name__
'fibo'

>>> fibo.__doc__
'Module for the computation of Fibonacci series'

>>> fibo.__file__
'/home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py'

>>> fibo.MAXFIBO
1000

>>> fibo.fibo_print(123)
Fibonacci series up to 123:
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Modules -2

To use a module it must be **imported**

How to use the module's **fibo()** function

```
>>> import fibo

>>> fibo.fibo(400)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]

>>> dir(fibo)
['MAXFIBO', '__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__', 'fibo', 'fibo_print']

>>> help(fibo)
Help on module fibo:

NAME
    fibo - Module for the computation of Fibonacci series

FUNCTIONS
    fibo(n)
        Returns the Fibonacci series up to n

    fibo_print(n)
        prints the Fibonacci series up to n

DATA
    MAXFIBO = 1000

FILE
    /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py

>>> fibo.__name__
'fibo'

>>> fibo.__doc__
'Module for the computation of Fibonacci series'

>>> fibo.__file__
'/home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py'

>>> fibo.MAXFIBO
1000

>>> fibo.fibo_print(123)
Fibonacci series up to 123:
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Modules -2

To use a module it must be **imported**

```
>>> import fibo
```

How to use the module's **fibo()** function

```
>>> fibo.fibo(400)
[1, 1, 2, 3, 5, 8,
```

The built-in function **dir()** shows module's content.

```
>>> dir(fibo)
['MAXFIBO', '__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__', 'fibo', 'fibo_print']

>>> help(fibo)
Help on module fibo:

NAME
    fibo - Module for the computation of Fibonacci series

FUNCTIONS
    fibo(n)
        Returns the Fibonacci series up to n

    fibo_print(n)
        prints the Fibonacci series up to n

DATA
    MAXFIBO = 1000

FILE
    /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py

>>> fibo.__name__
'fibo'

>>> fibo.__doc__
'Module for the computation of Fibonacci series'

>>> fibo.__file__
'/home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py'

>>> fibo.MAXFIBO
1000

>>> fibo.fibo_print(123)
Fibonacci series up to 123:
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Modules -2

To use a module it must be **imported**

How to use the module's **fibo()** function

The built-in function **dir()** shows module's content.

The built-in function **help()** prints in "document" format the content of some internal variables and other pieces of information derived from the module.

```
>>> import fibo

>>> fibo.fibo(400)
[1, 1, 2, 3, 5, 8,

>>> dir(fibo)
['MAXFIBO', '__built
'__loader__', '__nam

>>> help(fibo)
Help on module fibo:

NAME
    fibo - Module for the computation of Fibonacci series

FUNCTIONS
    fibo(n)
        Returns the Fibonacci series up to n

    fibo_print(n)
        prints the Fibonacci series up to n

DATA
    MAXFIBO = 1000

FILE
    /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py

>>> fibo.__name__
'fibo'

>>> fibo.__doc__
'Module for the computation of Fibonacci series'

>>> fibo.__file__
'/home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py'

>>> fibo.MAXFIBO
1000

>>> fibo.fibo_print(123)
Fibonacci series up to 123:
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Modules -2

```
>>> import fibo
```

To use a module it must be **imported**

```
>>> fibo.fibo(400)
[1, 1, 2, 3, 5, 8,
```

How to use the module's **fibo()** function

```
>>> dir(fibo)
['MAXFIBO', '__builtins
'__loader__', '__nam
```

The built-in function **dir()** shows module's content.

```
>>> help(fibo)
Help on module fibo:
```

The built-in function **help()** prints in "document" format the content of some internal variables and other pieces of information derived from the module.

```
NAME
    fibo - Module for the computation of Fibonacci series

FUNCTIONS
    fibo(n)
        Returns the Fibonacci series up to n

    fibo_print(n)
        prints the Fibonacci series up to n

DATA
    MAXFIBO = 1000

FILE
    /home/lfini/Personale

>>> fibo.__name__
'fibo'
```

The internal variable **__name__** holds the module's name (because the module has been imported, see also previous slide)

```
>>> fibo.__doc__
'Module for the computation of Fibonacci series'

>>> fibo.__file__
'/home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py'

>>> fibo.MAXFIBO
1000

>>> fibo.fibo_print(123)
Fibonacci series up to 123:
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Modules -2

```
>>> import fibo
```

To use a module it must be **imported**

```
>>> fibo.fibo(400)
[1, 1, 2, 3, 5, 8,
```

How to use the module’s **fibo()** function

```
>>> dir(fibo)
['MAXFIBO', '__builtins
'__loader__', '__nam
```

The built-in function **dir()** shows module’s content.

```
>>> help(fibo)
Help on module fibo:
```

The built-in function **help()** prints in “document” format the content of some internal variables and other pieces of information derived from the module.

```
NAME
    fibo - Module for the computation of Fibonacci series

FUNCTIONS
    fibo(n)
        Returns the Fibonacci series up to n

    fibo_print(n)
        prints the Fibonacci series up to n

DATA
    MAXFIBO = 1000

FILE
    /home/lfini/Personal

>>> fibo.__name__
'fibo'
```

The internal variable **__name__** holds the module’s name (because the module has been imported, see also previous slide)

```
>>> fibo.__doc__
'Module for the computa
```

The internal variable **__doc__** holds the string which is at the top of source file.

```
>>> fibo.__file__
'/home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py'

>>> fibo.MAXFIBO
1000

>>> fibo.fibo_print(123)
Fibonacci series up to 123:
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Modules -2

```
>>> import fibo
```

To use a module it must be **imported**

```
>>> fibo.fibo(400)
[1, 1, 2, 3, 5, 8,
```

How to use the module's **fibo()** function

```
>>> dir(fibo)
['MAXFIBO', '__builtins
'__loader__', '__nam
```

The built-in function **dir()** shows module's content.

```
>>> help(fibo)
Help on module fibo:
```

The built-in function **help()** prints in "document" format the content of some internal variables and other pieces of information derived from the module.

```
NAME
    fibo - Module for the computation of Fibonacci series

FUNCTIONS
    fibo(n)
        Returns the Fibonacci series up to n

    fibo_print(n)
        prints the Fibonacci series up to n

DATA
    MAXFIBO = 1000

FILE
    /home/lfini/Personale

>>> fibo.__name__
'fibo'
```

The internal variable **__name__** holds the module's name (because the module has been imported, see also previous slide)

```
>>> fibo.__doc__
'Module for the computa
```

The internal variable **__doc__** holds the string which is at the top of source file.

```
>>> fibo.__file__
'/home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py
```

The internal variable **__file__** holds the source file path

```
>>> fibo.MAXFIBO
1000

>>> fibo.fibo_print(123)
Fibonacci series up to 123:
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Modules -2

```
>>> import fibo
```

To use a module it must be **imported**

```
>>> fibo.fibo(400)
[1, 1, 2, 3, 5, 8,
```

How to use the module's **fibo()** function

```
>>> dir(fibo)
['MAXFIBO', '__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__nam
```

The built-in function **dir()** shows module's content.

```
>>> help(fibo)
Help on module fibo:

NAME
    fibo - Module for the computation of Fibonacci series

FUNCTIONS
    fibo(n)
        Returns the Fibonacci series up to n

    fibo_print(n)
        prints the Fibonacci series up to n

DATA
    MAXFIBO = 1000

FILE
    /home/lfini/Personale
```

The built-in function **help()** prints in "document" format the content of some internal variables and other pieces of information derived from the module.

```
>>> fibo.__name__
'fibo'
```

The internal variable __**name**__ holds the module's name (because the module has been imported, see also previous slide)

```
>>> fibo.__doc__
'Module for the computat
```

The internal variable __**doc**__ holds the string which is at the top of source file.

```
>>> fibo.__file__
'/home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py
```

The internal variable __**file**__ holds the source file path

```
>>> fibo.MAXFIBO
1000
```

Getting variable **MAXFIBO**

```
>>> fibo.fibo_print(123)
Fibonacci series up to 123:
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Modules -2

```
>>> import fibo

>>> fibo.fibo(400)
[1, 1, 2, 3, 5, 8,

>>> dir(fibo)
['MAXFIBO', '__builtins
'__loader__', '__nam

>>> help(fibo)
Help on module fibo:

NAME
    fibo - Module for the computation of Fibonacci series

FUNCTIONS
    fibo(n)
        Returns the Fibonacci series up to n

    fibo_print(n)
        prints the Fibonacci series up to n

DATA
    MAXFIBO = 1000

FILE
    /home/lfini/Personale

>>> fibo.__name__
'fibo'

>>> fibo.__doc__
'Module for the computa

>>> fibo.__file__
'/home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py

>>> fibo.MAXFIBO
1000

>>> fibo.fibo_print(123)
Fibonacci series up to 123:
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

To use a module it must be **imported**

How to use the module's **fibo()** function

The built-in function **dir()** shows module's content.

The built-in function **help()** prints in "document" format the content of some internal variables and other pieces of information derived from the module.

The internal variable __**name**__ holds the module's name (because the module has been imported, see also previous slide)

The internal variable __**doc**__ holds the string which is at the top of source file.

The internal variable __**file**__ holds the source file path

Getting variable **MAXFIBO**

How to use the **fibo_print()** function

namespaces and *scope* -1

- A *namespace* is a “container” for names.

namespaces and *scope* -1

- A *namespace* is a “container” for names.
- In a program there is a hierarchy of namespaces.

namespaces and *scope* -1

- A *namespace* is a “container” for names.
- In a program there is a hierarchy of namespaces.

file: `prcube.py`

```
note = "The cube of %d is: %d"

cube = 3.1415926

def print_cube(n):
    cube = n*n*n
    print(note % (n, cube))
```

namespaces and *scope* -1

- A *namespace* is a “container” for names.
- In a program there is a hierarchy of namespaces.

file: `prcube.py`

```
note = "The cube of %d is: %d"

cube = 3.1415926

def print_cube(n):
    cube = n*n*n
    print(note % (n, cube))
```

namespaces and *scope* -1

- A *namespace* is a “container” for names.
- In a program there is a hierarchy of namespaces.

file: `prcube.py` ⟸

Module `prcube` is a namespace containing names: **note**, **cube**, **print_cube**.

```
note = "The cube of %d is: %d"

cube = 3.1415926

def print_cube(n):
    cube = n*n*n
    print(note % (n, cube))
```

namespaces and *scope* -1

- A *namespace* is a “container” for names.
- In a program there is a hierarchy of namespaces.

file: `prcube.py`

```
note = "The cube of %d is: %d"

cube = 3.1415926

def print_cube(n):
    cube = n*n*n
    print(note % (n, cube))
```

Module `prcube` is a namespace containing names: **note**, **cube**, **print_cube**.

The function `print_cube()` is another namespace containing names **cube** ed **n**

namespaces and *scope* -1

- A *namespace* is a “container” for names.
- In a program there is a hierarchy of namespaces.

file: `prcube.py`

```
note = "The cube of %d is: %d"

cube = 3.1415926

def print_cube(n):
    cube = n*n*n
    print(note % (n, cube))
```

Module `prcube` is a namespace containing names: **note**, **cube**, **print_cube**.

The function `print_cube()` is
a
n

Note: the local namespace of any function is not visible outside the function

namespaces and *scope* -1

- A *namespace* is a “container” for names.
- In a program there is a hierarchy of namespaces.

file: `prcube.py`

```
note = "The cube of %d is: %d"

cube = 3.1415926

def print_cube(n):
    cube = n*n*n
    print(note % (n, cube))
```

Module `prcube` is a namespace containing names: **note**, **cube**, **print_cube**.

The function `print_cube()` is
a
n

Note: the local namespace of any function is not visible outside the function

Let’s launch the Python interpreter:

```
$ python
Python 3.5.2 (default, Nov 23 2017, 16:37:01)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.

>>> cube = "The third power"

>>> import prcube

>>> prcube.print_cube(11)
The cube of 11 is: 1331

>>> cube
'The third power'

>>> prcube.cube
3.1415926
```

namespaces and *scope* -1

- A *namespace* is a “container” for names.
- In a program there is a hierarchy of namespaces.

file: `prcube.py`

Module `prcube` is a namespace containing names: **note**, **cube**, **print_cube**.

```
note = "The cube of %d is: %d"

cube = 3.1415926

def print_cube(n):
    cube = n*n*n
    print(note % (n, cube))
```

The function print_cube() is a
n

Note: the local namespace of any function is not visible outside the function

Let’s launch the Python interpreter:

The Python interpreter is the level 0 namespace

```
$ python
Python 3.5.2 (default, Nov 23 20
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credi                                ation.

>>> cube = "The third power"

>>> import prcube

>>> prcube.print_cube(11)
The cube of 11 is: 1331

>>> cube
'The third power'

>>> prcube.cube
3.1415926
```

namespaces and *scope* -1

- A *namespace* is a “container” for names.
- In a program there is a hierarchy of namespaces.

file: `prcube.py`

Module `prcube` is a namespace containing names: **note**, **cube**, **print_cube**.

```
note = "The cube of %d is: %d"

cube = 3.1415926

def print_cube(n):
    cube = n*n*n
    print(note % (n, cube))
```

The function print_cube() is
a
n

Note: the local namespace of any function is not visible outside the function

Let’s launch the Python interpreter:

```
$ python
Python 3.5.2 (default, Nov 23 20
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credi

>>> cube = "The third power"

>>> import prcube

>>> prcube.print_cube(11)
The cube of 11 is: 1331

>>> cube
'The third power'

>>> prcube.cube
3.1415926
```

The Python interpreter is

The name **cube** is created in level 0 namespace

namespaces and *scope* -1

- A *namespace* is a “container” for names.
- In a program there is a hierarchy of namespaces.

file: `prcube.py`

Module `prcube` is a namespace containing names: **note**, **cube**, **print_cube**.

```
note = "The cube of %d is: %d"

cube = 3.1415926

def print_cube(n):
    cube = n*n*n
    print(note % (n, cube))
```

The function print_cube() is
a
n

Note: the local namespace of any function is not visible outside the function

Let’s launch the Python interpreter:

```
$ python
Python 3.5.2 (default, Nov 23 20
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credi

>>> cube = "The third po

>>> import prcube

>>> prcube.print_cube(11
The cube of 11 is: 1331

>>> cube
'The third power'

>>> prcube.cube
3.1415926
```

The Python interpreter is

The name **cube** is created in

The **import** statement creates the name **prcube** referring to the full namespace of the `prcube` module.

namespaces and *scope* -1

- A *namespace* is a “container” for names.
- In a program there is a hierarchy of namespaces.

file: `prcube.py`

```
note = "The cube of %d is: %d"

cube = 3.1415926

def print_cube(n):
    cube = n*n*n
    print(note % (n, cube))
```

Module `prcube` is a namespace containing names: **note**, **cube**, **print_cube**.

The function `print_cube()` is
a
n

Note: the local namespace of any function is not visible outside the function

Let’s launch the Python interpreter:

```
$ python
Python 3.5.2 (default, Nov 23 20
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credi

>>> cube = "The third po

>>> import prcube

>>> prcube.print_cube(11)
The cube of 11 is: 1331

>>> cube
'The third power'

>>> prcube.cube
3.1415926
```

The Python interpreter is

The name **cube** is created in level 0 namespace

The **import** statement creates the name **prcube** referring to the full namespace of the `prcube` module.

The **print_cube()** function defines another namespace, containing a variable named **cube**

namespaces and *scope* -1

- A *namespace* is a “container” for names.
- In a program there is a hierarchy of namespaces.

file: `prcube.py`

Module `prcube` is a namespace containing names: **note**, **cube**, **print_cube**.

```
note = "The cube of %d is: %d"

cube = 3.1415926

def print_cube(n):
    cube = n*n*n
    print(note % (n, cube))
```

The function print_cube() is

Note: the local namespace of any function is not visible outside the function

Let’s launch the Python interpreter:

```
$ python
Python 3.5.2 (default, Nov 23 20
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credi

>>> cube = "The third po

>>> import prcube

>>> prcube.print_cube(11
The cube of 11 is: 1331

>>> cube
'The third power'

>>> prcube.cube
3.1415926
```

The Python interpreter is

The name **cube** is created in level 0 namespace

The **import** statement creates the name **prcube** referring to the full namespace of the `prcube` module.

The **print_cube()** function defines another namespace, containing a variable named **cube**

In the above example the name **cube** shows up three times in three different namespaces, i.e.: referring to three different objects.

namespaces and *scope* -2

- ***Scope***: All the “area” where a name is referred to the same object.
- Scope rules:

namespaces and *scope* -2

- ***Scope***: All the “area” where a name is referred to the same object.
- Scope rules:
    - Name creation: in the currently active namespace.

namespaces and *scope* -2

- ***Scope***: All the “area” where a name is referred to the same object.
- Scope rules:
  - Name creation: in the currently active namespace.
  - Name search: starting from currently active namespace and up the hierarchy

namespaces and *scope* -2

- ***Scope***: All the “area” where a name is referred to the same object.
- Scope rules:
  - Name creation: in the currently active namespace.
  - Name search: starting from currently active namespace and up the hierarchy

Let’s go back to the previous example:

```
note = "The cube of %d is: %d"

cube = 3.1415926

def print_cube(n):

    cube = n*n*n

    print(note % (n, cube))

....
```

namespaces and *scope* -2

- ***Scope***: All the “area” where a name is referred to the same object.
- Scope rules:
  - Name creation: in the currently active namespace.
  - Name search: starting from currently active namespace and up the hierarchy

Let’s go back to the previous example:

```
note = "The cube of %d is: %d"

cube = 3.1415926

def print_cube(n):

    cube = n*n*n

    print(note % (n, cube))

....
```

namespaces and *scope* -2

- ***Scope***: All the “area” where a name is referred to the same object.
- Scope rules:
  - Name creation: in the currently active namespace.
  - Name search: starting from currently active namespace and up the hierarchy

Let’s go back to the prev

The name **note** is created here

```
note = "The cube of %d is: %d"

cube = 3.1415926

def print_cube(n):

    cube = n*n*n

    print(note % (n, cube))

....
```

namespaces and *scope* -2

- ***Scope***: All the “area” where a name is referred to the same object.
- Scope rules:
  - Name creation: in the currently active namespace.
  - Name search: starting from currently active namespace and up the hierarchy

Let’s go back to the prev

The name **note** is created here

The name **cube**$_1$ is created here

```
note = "The cube of %d is: %d"

cube = 3.1415926

def print_cube(n):

    cube = n*n*n

    print(note % (n, cube))

....
```

namespaces and *scope* -2

- ***Scope***: All the "area" where a name is referred to the same object.
- Scope rules:
  - Name creation: in the currently active namespace.
  - Name search: starting from currently active namespace and up the hierarchy

Let's go back to the prev

```
note = "The cube of %d is: %d"

cube = 3.1415926

def print_cube(n):

    cube = n*n*n

    print(note % (n, cube))

....
```

The name **note** is created here

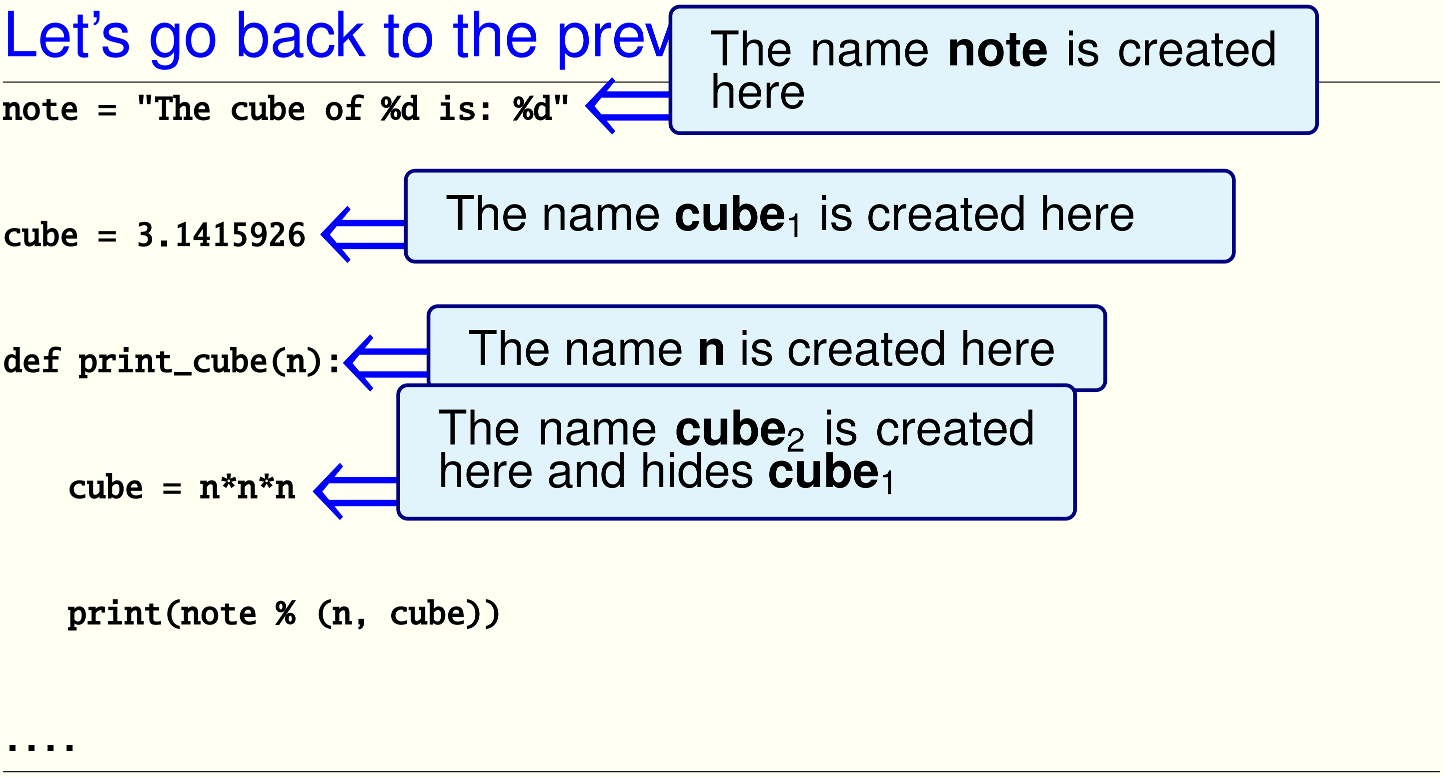
The name **cube**$_1$ is created here

The name **n** is created here

namespaces and *scope* -2

- ***Scope***: All the “area” where a name is referred to the same object.
- Scope rules:
    - Name creation: in the currently active namespace.
    - Name search: starting from currently active namespace and up the hierarchy

Let’s go back to the prev



```
note = "The cube of %d is: %d"

cube = 3.1415926

def print_cube(n):

    cube = n*n*n

    print(note % (n, cube))

....
```

The name **note** is created here

The name **cube**$_1$ is created here

The name **n** is created here

The name **cube**$_2$ is created here and hides **cube**$_1$

namespaces and *scope* -2

- ***Scope***: All the “area” where a name is referred to the same object.
- Scope rules:
  - Name creation: in the currently active namespace.
  - Name search: starting from currently active namespace and up the hierarchy

Let’s go back to the prev

```
note = "The cube of %d is: %d"

cube = 3.1415926

def print_cube(n):

    cube = n*n*n

    print(note % (n, cube))

....
```

The name **note** is created here

The name **cube**$_1$ is created here

The name **n** is created here

The na
here ar

Here the name **note** is searched up the hierarchy and found one step above. **n** and **cube** are found in the current namespace

namespaces and *scope* -2

- ***Scope***: All the “area” where a name is referred to the same object.
- Scope rules:
    - Name creation: in the currently active namespace.
    - Name search: starting from currently active namespace and up the hierarchy

Let’s go back to the prev

```
note = "The cube of %d is: %d"

cube = 3.1415926

def print_cube(n):

    cube = n*n*n

    print(note % (

....
```

The name **note** is created here

The name **cube**$_1$ is created here

The name **n** is created here

The na
here ar

Here the name **note** is searched up the hierarchy and found one step above. **n** and **cube** are found in the current namespace

Here the name **cube**$_2$ disappears and **cube** refers again to the object created at the second line of the file

namespaces and scope -2

- ***Scope***: All the “area” where a name is referred to the same object.
- Scope rules:
  - Name creation: in the currently active namespace.
  - Name search: starting from currently active namespace and up the hierarchy

Let’s go back to the prev

```
note = "The cube of %d is: %d"

cube = 3.1415926

def print_cube(n):

    cube = n*n*n

    print(note % (

....
```

The name **note** is created here

The name **cube**$_1$ is created here

The name **n** is created here

The na
here ar

Here the name **note** is searched up the hierarchy and found one step above. **n** and **cube** are found in the

Here the name **cube**$_2$ disappears and **cube** refers again to the object created at the second line of the file

→

global namespace -1

A well known, widely used, debugging technique consists in adding print function calls where needed, to show variable values as an help in understanding the sequence of operations.

global namespace -1

A well known, widely used, debugging technique consists in adding print function calls where needed, to show variable values as an help in understanding the sequence of operations.

To avoid the need to remove print functions from the code when finished testing, it is common to place the debugging statements under a conditional statement as in the following example.

global namespace -1

A well known, widely used, debugging technique consists in adding print function calls where needed, to show variable values as an help in understanding the sequence of operations.

To avoid the need to remove print functions from the code when finished testing, it is common to place the debugging statements under a conditional statement as in the following example.

file: scope1.py

```
DEBUG = True

def main():
    print("Program starting ...")
    if DEBUG:
        print("Running in debug mode")

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

file: scope2.py

```
DEBUG = False

def main():
    print("Program starting ...")
    if DEBUG:
        print("Running in debug mode")

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

global namespace -1

A well known, widely used, debugging technique consists in adding print function calls where needed, to show variable values as an help in understanding the sequence of operations.

To avoid the need to remove print functions from the code when finished testing, it is common to place the debugging statements under a conditional statement as in the following example.

file: scope1.py

```
DEBUG = True

def main():
    print("Program starting ...")
    if DEBUG:
        print("Running in debug mode")

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

file: scope2.py

```
DEBUG = False

def main():
    print("Program starting ...")
    if DEBUG:
        print("Running in debug mode")

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

global namespace -1

A well known, widely used, debugging technique consists in adding print function calls where needed, to show variable values as an help in understanding the sequence of operations.

To avoid the need to remove print functions from the code when finished testing, it is common to place the debugging statements under a conditional statement as in the following example.

file: scope1.py

```
DEBUG = True ⟸

def main():
    print("Program starting ...")
    if DEBUG:
        print("Running in debug mode")

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

`scope1.py` e `scope2.py` are identical except for the value of variable `DEBUG`.

file: scope2.py

```
DEBUG = False ⟸

def main():
    print("Program starting ...")
    if DEBUG:
        print("Running in debug mode")

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

global namespace -1

A well known, widely used, debugging technique consists in adding print function calls where needed, to show variable values as an help in understanding the sequence of operations.

To avoid the need to remove print functions from the code when finished testing, it is common to place the debugging statements under a conditional statement as in the following example.

file: scope1.py

```
DEBUG = True ⇐

def main():
    print("Program starting ...")
    if DEBUG:
        print("Running in debug mode")

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

`scope1.py` e `scope2.py` are identical except for the value of variable `DEBUG`.

file: scope2.py

```
DEBUG = False ⇐

def main():
    print("Program starting ...")
    if DEBUG:
        print("Running in debug mode")

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

```
$ python scope1.py
Program starting ...
Running in debug mode

$ python scope2.py
Program starting ...
```

global namespace - 2

Usually it is better to be able to select the running mode at program execution without the need to modify the `DEBUG` variable, e.g.: by using an optional parameter on the command line.

global namespace - 2

Usually it is better to be able to select the running mode at program execution without the need to modify the `DEBUG` variable, e.g.: by using an optional parameter on the command line.

file: scope3.py

```
import sys

DEBUG = False

def main():
    if "-d" in sys.argv:
        DEBUG = True

    print("Program starting ...")
    if DEBUG:
        print("Running in debug mode")

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

global namespace - 2

Usually it is better to be able to select the running mode at program execution without the need to modify the `DEBUG` variable, e.g.: by using an optional parameter on the command line.

file: scope3.py

```
import sys

DEBUG = False

def main():
    if "-d" in sys.argv:
        DEBUG = True

    print("Program starting ...")
    if DEBUG:
        print("Running in debug mode")

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

global namespace - 2

Usually it is better to be able to select the running mode at program execution without the need to modify the `DEBUG` variable, e.g.: by using an optional parameter on the command line.

file: scope3.py

```
import sys

DEBUG = False

def main():
    if "-d" in sys.argv:
        DEBUG = True

    print("Program starting ...")
    if DEBUG:
        print("Running in debug mode")

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

The **sys.argv** variable holds the command string, split into “words”

global namespace - 2

Usually it is better to be able to select the running mode at program execution without the need to modify the `DEBUG` variable, e.g.: by using an optional parameter on the command line.

file: scope3.py

```
import sys

DEBUG = False

def main():
    if "-d" in sys.argv:
        DEBUG = True

    print("Program starting ...")
    if DEBUG:
        print("Running in debug mode")

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

The **sys.argv** variable holds the command string, split into “words”

Let’s run the program:

global namespace - 2

Usually it is better to be able to select the running mode at program execution without the need to modify the `DEBUG` variable, e.g.: by using an optional parameter on the command line.

file: scope3.py

```
import sys

DEBUG = False

def main():
    if "-d" in sys.argv:
        DEBUG = True

    print("Program starting ...")
    if DEBUG:
        print("Running in debug mode")

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

The **sys.argv** variable holds the command string, split into “words”

Let’s run the program:

```
$ python scope3.py -d
Program starting ...
Running in debug mode
```

global namespace - 2

Usually it is better to be able to select the running mode at program execution without the need to modify the `DEBUG` variable, e.g.: by using an optional parameter on the command line.

file: scope3.py

```
import sys

DEBUG = False

def main():
    if "-d" in sys.argv:
        DEBUG = True

    print("Program starting ...")
    if DEBUG:
        print("Running in debug mode")

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

⇐ The **sys.argv** variable holds the command string, split into “words”

Let’s run the program:

```
$ python scope3.py -d
Program starting ...
Running in debug mode
```

⇐ 1. Adding option **-d**

global namespace - 2

Usually it is better to be able to select the running mode at program execution without the need to modify the `DEBUG` variable, e.g.: by using an optional parameter on the command line.

file: scope3.py

```
import sys

DEBUG = False

def main():
    if "-d" in sys.argv:
        DEBUG = True

    print("Program starting ...")
    if DEBUG:
        print("Running in debug mode")

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

The **sys.argv** variable holds the command string, split into “words”

Let’s run the program:

```
$ python scope3.py -d
Program starting ...
Running in debug mode
```

1. Adding option **-d**

```
$ python scope3.py
Program starting ...
Traceback (most recent call last):
  File "scope3.py", line 14, in <module>
    main()
  File "scope3.py", line 10, in main
    if DEBUG:
UnboundLocalError: local variable 'DEBUG' referenced before assignment
```

global namespace - 2

Usually it is better to be able to select the running mode at program execution without the need to modify the `DEBUG` variable, e.g.: by using an optional parameter on the command line.

file: scope3.py

```
import sys

DEBUG = False

def main():
    if "-d" in sys.argv:
        DEBUG = True

    print("Program starting ...")
    if DEBUG:
        print("Running in debug mode")

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

The **sys.argv** variable holds the command string, split into “words”

Let’s run the program:

```
$ python scope3.py -d
Program starting ...
Running in debug mode
```

1. Adding option **-d**

```
$ python scope3.py
Program starting ...
Traceback (most recent call last):
  File "scope3.py", line 14, in <module>
    main()
  File "scope3.py", line 10, in main
    if DEBUG:
UnboundLocalError: local variable ’DEBUG’ referenced before assignment
```

2. Without option **-d**

global namespace - 2

Usually it is better to be able to select the running mode at program execution without the need to modify the `DEBUG` variable, e.g.: by using an optional parameter on the command line.

file: scope3.py

```python
import sys

DEBUG = False

def main():
    if "-d" in sys.argv:
        DEBUG = True

    print("Program starting ...")
    if DEBUG:
        print("Running in debug mode")

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

The **sys.argv** variable holds the command string, split into “words”

Let’s run the program:

```
$ python scope3.py -d
Program starting ...
Running in debug mode
```

1. Adding option **-d**

```
$ python scope3.py
Program starting ...
Traceback (most recent call last):
  File "scope3.py", line 14, in <module>
    main()
  File "scope3.py", line 10, in main
    if DEBUG:
UnboundLocalError: local variable ’DEBUG
```

2. Without option **-d**

Note the error at line 10.

The variable `DEBUG` is not defined!

global namespace - 2

Usually it is better to be able to select the running mode at program execution without the need to modify the `DEBUG` variable, e.g.: by using an optional parameter on the command line.

file: scope3.py

```
import sys

DEBUG = False

def main():
    if "-d" in sys.argv:
        DEBUG = True

    print("Program starting ...")
    if DEBUG:
        print("Running in debug mode")

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

The **sys.argv** variable holds the command string, split into “words”

Let’s run the program:

```
$ python scope3.py -d
Program starting ...
Running in debug mode
```

1. Adding option **-d**

2. Without option **-d**

```
$ python scope3.py
Program starting ...
Traceback (most recent call last):
  File "scope3.py", line 14, in <module>
    main()
  File "scope3.py", line 10, in main
    if DEBUG:
UnboundLocalError: local variable ’DEBUG
```

Note the error at line 10.

The variable `DEBUG` is not defined!

→

global namespace -3

Let’s go back to scope rules:

- in `scope3.py` we have the assignment:
  `DEBUG = True`

global namespace -3

Let’s go back to scope rules:

- in `scope3.py` we have the assignment:
  `DEBUG = True`
- the name `DEBUG` is thus located in the *local* namespace of function `main()`

global namespace -3

Let’s go back to scope rules:

- in `scope3.py` we have the assignment:
  `DEBUG = True`
- the name `DEBUG` is thus located in the *local* namespace of function `main()`
- but the variable is actually created **only** if the conditional part of the first `if` statement is executed.

global namespace -3

Let’s go back to scope rules:

- in `scope3.py` we have the assignment:
  `DEBUG = True`
- the name `DEBUG` is thus located in the *local* namespace of function `main()`
- but the variable is actually created **only** if the conditional part of the first `if` statement is executed.
- Note: also when variable `DEBUG` is created it only lives within the `main()` function: variable `DEBUG` defined at line 2 is not affected!

global namespace -3

Let's go back to scope rules:

- in `scope3.py` we have the assignment:
  `DEBUG = True`
- the name `DEBUG` is thus located in the *local* namespace of function `main()`
- but the variable is actually created **only** if the conditional part of the first `if` statement is executed.
- Note: also when variable `DEBUG` is created it only lives within the `main()` function: variable `DEBUG` defined at line 2 is not affected!

file: scope4.py – Working version

```
import sys

DEBUG = False

def main():
    global DEBUG
    if "-d" in sys.argv:
        DEBUG = True

    print("Program starting ...")
    if DEBUG:
        print("Running in debug mode")

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

global namespace -3

Let's go back to scope rules:

- in `scope3.py` we have the assignment:
  `DEBUG = True`
- the name `DEBUG` is thus located in the *local* namespace of function `main()`
- but the variable is actually created **only** if the conditional part of the first `if` statement is executed.
- Note: also when variable `DEBUG` is created it only lives within the `main()` function: variable `DEBUG` defined at line 2 is not affected!

file: scope4.py – Working version

```
import sys

DEBUG = False

def main():
    global DEBUG
    if "-d" in sys.argv:
        DEBUG = True

    print("Program starting ...")
    if DEBUG:
        print("Running in debug mode")

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

global namespace -3

Let’s go back to scope rules:

- in `scope3.py` we have the assignment:
  `DEBUG = True`
- the name `DEBUG` is thus located in the *local* namespace of function `main()`
- but the variable is actually created **only** if the conditional part of the first `if` statement is executed.
- Note: also when variable `DEBUG` is created it only lives within the `main()` function: variable `DEBUG` defined at line 2 is not affected!

file: scope4.py – Working version

```
import sys

DEBUG = False

def main():
    global DEBUG ⟸
    if "-d" in sys.argv:
        DEBUG = True

    print("Program starting ...")
    if DEBUG:
        print("Running in debug mode")

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

> The `global DEBUG` statement forces python to create (or search) the name `DEBUG` into the global namespace, i.e.: the topmost in the namespace hierarchy.

Classes and Objects

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 __str__(self):
        return self.name

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

Classes and Objects

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 __str__(self):
        return self.name

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

Classes and Objects

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

file: number.py

Definition of class **Number**

```
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 __str__(self):
        return self.name

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

Classes and Objects

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

file: number.py

Definition of class **Number**

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

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

    def __str__(self):
        return self.name

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

names: class attribute

Classes and Objects

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

file: number.py

Definition of class **Number**

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

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

    def __str__(self):
        return self.name

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

names: class attribute

Special method `__init__()`: **constructor**

Classes and Objects

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'
         'five', 'six','seven

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

   def __str__(self):
       return self.name

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

Definition of class **Number**

names: class attribute

Special method `__init__()`: **constructor**

Instance attributes

Classes and Objects

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

file: number.py

Definition of class **Number**

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

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

    def __str__(self):
        return self.name

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

names: class attribute

Special method `__init__()`: **constructor**

Method: `__str__()`; *overloaded* because it is predefined in every class

Classes and Objects

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

file: number.py

Definition of class **Number**

```
class Number:
    "An example of class: Number()"
    names=('zero','one','two','three'
           'five', 'six','seven
    
    def __init__(self, n):
       self.name=Number.names[n]
       self.value=n
    
    def __str__(self):
        return self.name
    
    def upper(self):
        return self.name.upp
```

names: class attribute

Special method `__init__()`: **constructor**

Method: `__str__()`; *overloaded* because it is predefined in every class

Standard user defined method: `upper()`

Classes and Objects

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

file: number.py

Definition of class **Number**

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

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

    def __str__(self):
        return self.name

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

names: class attribute

Special method `__init__()`: **constructor**

Method: `__str__()`; *overloaded* because it is predefined

Stand
upper

Note: in all methods the first argument (required) refers to the instance (it is usually named `self`)

Classes and Objects

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

file: number.py

Definition of class **Number**

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

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

    def __str__(self):
        return self.name

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

names: class attribute

Special method `__init__()`: **constructor**

Method: `__str__()`; *overloaded* because it is predefined

Stand
upper

Note: in all methods the first argument (required) refers to the instance (it is usually named `self`)

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>
>>> print(a, b)
two three
>>> a.value
2
>>> a.upper()
'TWO'
>>>
```

Classes and Objects

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

file: number.py

Definition of class **Number**

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

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

    def __str__(self):
        return self.name

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

names: class attribute

Special method `__init__()`: **constructor**

Method: `__str__()`; *overloaded* because it is predefined

Stand
upper

Note: in all methods the first argument (required) refers to the instance (it is usually named `self`)

How to use class `Number`:

```
>>> from number impor
>>> a=Number(2)
>>> b=Number(3)
>>> a
<number.Number object at 0x7f9153a695f8>
>>> b
<number.Number object at 0x7f9153a695c0>
>>> print(a, b)
two three
>>> a.value
2
>>> a.upper()
'TWO'
>>>
```

Creating two instances (objects) of class `Number`

Classes and Objects

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

file: number.py

Definition of class **Number**

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

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

    def __str__(self):
        return self.name

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

names: class attribute

Special method `__init__()`: **constructor**

Method: `__str__()`; *overloaded* because it is predefined

Stand upper

Note: in all methods the first argument (required) refers to the instance (it is usually named `self`)

How to use class `Number`:

```
>>> from number impor
>>> a=Number(2)
>>> b=Number(3)
>>> a
<number.Number object at 0x7f9153a695f8>
>>> b
<number.Number object
>>> print(a, b)
two three
>>> a.value
2
>>> a.upper()
'TWO'
>>>
```

Creating two instances (objects) of class `Number`

The `print()` function uses the object's method `__str__()` implicitly

Classes and Objects

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

file: number.py

Definition of class **Number**

```
class Number:
    "An example of class: Number()"
    names=('zero','one','two'
           'five', 'six','seve

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

    def __str__(self):
        return self.name

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

names: class attribute

Special method `__init__()`: **constructor**

Method: `__str__()`; *overloaded* because it is predefined

Stanc
uppe

Note: in all methods the first argument (required) refers to the instance (it is usually named `self`)

How to use class `Number`:

```
>>> from number impor
>>> a=Number(2)
>>> b=Number(3)
>>> a
<number.Number object at 0x7f9153a695f8>
>>> b
<number.Number object
>>> print(a, b)
two three
>>> a.value
2
>>> a.upper()
'TWO'
>>>
```

Creating two instances (objects) of class `Number`

The `print()` function uses the object's method `__str__()` implicitly

Using object's attribute `value`

Classes and Objects

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

file: number.py

Definition of class **Number**

```
class Number:
    "An example of class: Number()"
    names=('zero','one','two' ,'thr
          'five', 'six','seven
    
    def __init__(self, n):
       self.name=Number.name
       self.value=n
    
    def __str__(self):
      return self.name
    
    def upper(self):
      return self.name.upp
```

names: class attribute

Special method `__init__()`: **constructor**

Method: `__str__()`; *overloaded* because it is predefined

Stand
upper

Note: in all methods the first argument (required) refers to the instance (it is usually named `self`)

How to use class `Number`:

```
>>> from number impor
>>> a=Number(2)
>>> b=Number(3)
>>> a
<number.Number object at 0x7f9153a695f8>
>>> b
<number.Number object
>>> print(a, b)
two three
>>> a.value
2
>>> a.upper()
'TWO'
>>>
```

Creating two instances (objects) of class `Number`

The `print()` function uses the object's method `__str__()` implicitly

Using object's attribute `value`

Using object's method `upper()`

Classes and Objects

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

file: number.py

Definition of class **Number**

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

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

    def __str__(self):
        return self.name

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

names: class attribute

Special method `__init__()`: **constructor**

Method: `__str__()`; *overloaded* because it is predefined

Stand
upper

Note: in all methods the first argument (required) refers to the instance (it is usually named `self`)

How to use class `Number`:

```
>>> from number impor
>>> a=Number(2)
>>> b=Number(3)
>>> a
<number.Number object at 0x7f9153a695f8>
>>> b
<number.Number object
>>> print(a, b)
two three
>>> a.value
2
>>> a.upper()
'TWO'
>>>
```

Creating two instances (objects) of class `Number`

The `print()` function uses the object's method `__str__()` implicitly

Using object's attribute `value`

Us

Note: in method call the first required argument is used implicitly.

`a.upper()` :: `Number.upper(a)`

Classes and Objects

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

file: number.py

Definition of class **Number**

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

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

    def __str__(self):
        return self.name

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

names: class attribute

Special method `__init__()`: **constructor**

Method: `__str__()`; *overloaded* because it is predefined

Stand
upper

Note: in all methods the first argument (required) refers to the instance (it is usually named `self`)

How to use class `Number`:

```
>>> from number impor
>>> a=Number(2)
>>> b=Number(3)
>>> a
<number.Number object at 0x7f9153a695f8>
>>> b
<number.Number object
>>> print(a, b)
two three
>>> a.value
2
>>> a.upper()
'TWO'
>>>
```

Creating two instances (objects) of class `Number`

The `print()` function uses the object's method `__str__()` implicitly

Using object's attribute `value`

Us

Note: in method call the first required argument is used implicitly.

`a.upper()` :: `Number.upper(a)`

→

Derived classes

Proceeding further with the previous example:

```
>>> a+b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'Number' and 'Number'
>>>
```

Derived classes

Proceeding further with the previous example:

```
>>> a+b
Traceback (most recent call las
  File "<stdin>", line 1, in <m
TypeError: unsupported operand                                    r'
>>>
```

Class `Number` does not provide an implementation for addition

Derived classes

Proceeding further with the previous example:

```
>>> a+b
Traceback (most recent call las
  File "<stdin>", line 1, in <m
TypeError: unsupported operand                              r',
>>>
```

Class `Number` does not provide an implementation for addition

file: `numberext.py`

```python
from number import Number

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

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

Derived classes

Proceeding further with the previous example:

```
>>> a+b
Traceback (most recent call las
  File "<stdin>", line 1, in <m
TypeError: unsupported operand                                  er'
>>>
```

Class `Number` does not provide an implementation for addition

file: `numberext.py`

```
from number import Number

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

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

Here we define a class *derived* from `Number`

Derived classes

Proceeding further with the previous example:

```
>>> a+b
Traceback (most recent call las
  File "<stdin>", line 1, in <m
TypeError: unsupported operand                                   er'
>>>
```

Class `Number` does not provide an implementation for addition

file: `numberext.py`

```
from number import Number

class NumberExt(Number):
    "Number with addition"
    def __add__(self, x):
        return NumberExt(self.v

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

Here we define a class *derived* from `Number`

Implementation of the special method: `__add__`

Derived classes

Proceeding further with the previous example:

```
>>> a+b
Traceback (most recent call las
  File "<stdin>", line 1, in <m
TypeError: unsupported operand                                    er'
>>>
```

Class `Number` does not provide an implementation for addition

file: `numberext.py`

```
from number import Number

class NumberExt(Number):
    "Number with addition"
    def __add__(self, x):
        return NumberExt(self.

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

Here we define a class *derived* from `Number`

Implementation of the special method: add

Re-implementation (*overload*) of method `upper()`

Derived classes

Proceeding further with the previous example:

```
>>> a+b
Traceback (most recent call las
  File "<stdin>", line 1, in <m
TypeError: unsupported operand                                  r'
>>>
```

Class `Number` does not provide an implementation for addition

file: `numberext.py`

```
from number import Number

class NumberExt(Number):
    "Number with addition"
    def __add__(self, x):
        return NumberExt(self.

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

Here we define a class *derived* from `Number`

Implementation of the special method: add

Re-implementation (*overload*) of method `upper()`

Let's use the new class:

```
>>> from numberext import NumberExt
>>> a=NumberExt(2)
>>> b=NumberExt(3)
>>> a
<numberext.NumberExt object at 0x7fe17970dc88>
>>> b
<numberext.NumberExt object at 0x7fe17970dc50>
>>> print(a, "+", b, "=", a+b)
two + three = five
>>> a.upper()
'Two'
>>>
```

Derived classes

Proceeding further with the previous example:

```
>>> a+b
Traceback (most recent call las
  File "<stdin>", line 1, in <m
TypeError: unsupported operand                                  er'
>>>
```

Class `Number` does not provide an implementation for addition

file: `numberext.py`

```
from number import Number

class NumberExt(Number):
    "Number with addition"
    def __add__(self, x):
        return NumberExt(self.

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

Here we define a class *derived* from `Number`

Implementation of the special method: add

Re-implementation (*overload*) of method `upper()`

Let's use the new class:

```
>>> from numberext impo
>>> a=NumberExt(2)
>>> b=NumberExt(3)
>>> a
<numberext.NumberExt object at 0x7fe17970dc88>
>>> b
<numberext.NumberExt object at 0x7fe17970dc50>
>>> print(a, "+", b, "=", a+b)
two + three = five
>>> a.upper()
'Two'
>>>
```

Creating two instances of class `NumberExt`

Derived classes

Proceeding further with the previous example:

```
>>> a+b
Traceback (most recent call las
  File "<stdin>", line 1, in <m
TypeError: unsupported operand                                  er'
>>>
```

Class `Number` does not provide an implementation for addition

file: `numberext.py`

```
from number import Number

class NumberExt(Number):
    "Number with addition"
    def __add__(self, x):
        return NumberExt(self.

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

Here we define a class *derived* from `Number`

Implementation of the special method: __add

Re-implementation (*overload*) of method `upper()`

Let's use the new class:

```
>>> from numberext impo
>>> a=NumberExt(2)
>>> b=NumberExt(3)
>>> a
<numberext.NumberExt object at 0x7fe17970dc88>
>>> b
<numberext.NumberExt object at 0x7fe17970dc50>
>>> print(a, "+", b, "=", a+b)
two + three = five
>>> a.upper()
'Two'
>>>
```

Creating two instances of class `NumberExt`

The addition operation (**+**) calls implicitly the `__add__()` method. The latter returns an instance of `NumberExt` class with value equal to the sum of the two addends

Derived classes

Proceeding further with the previous example:

```
>>> a+b
Traceback (most recent call las
  File "<stdin>", line 1, in <m
TypeError: unsupported operand
>>>
```

Class `Number` does not provide an implementation for addition

file: `numberext.py`

```
from number import Number

class NumberExt(Number):
    "Number with addition"
    def __add__(self, x):
        return NumberExt(self.

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

Here we define a class *derived* from `Number`

Implementation of the special method: __add

Re-implementation (*overload*) of method `upper()`

Let's use the new class:

```
>>> from numberext impo
>>> a=NumberExt(2)
>>> b=NumberExt(3)
>>> a
<numberext.NumberExt object at 0x7fe17970dc88>
>>> b
<numberext.NumberExt object at 0x7fe17970dc50>
>>> print(a, "+", b, "=", a+b)
two + three = five
>>> a.upper()
'Two'
>>>
```

Creating two instances of class `NumberExt`

The addition operation (**+**) calls implicitly the `__add__()` method. The latter returns an instance of `NumberExt` class with value equal to the sum of the two addends

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

Iterators and Generators -1

Iterators and **generators** are tools to generate sequences of objects in an efficient way.

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
```

Iterators and Generators -1

Iterators and **generators** are tools to generate sequences of objects in an efficient way.

file: fibo1.py

An `iterator` is a class ...

```
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
```

Iterators and Generators -1

Iterators and **generators** are tools to generate sequences of objects in an efficient way.

file: fibo1.py

An `iterator` is a class ...

... provided with an `__iter__` method,

```
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
```

Iterators and Generators -1

Iterators and **generators** are tools to generate sequences of objects in an efficient way.

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
```

An `iterator` is a class ...

... provided with an `__iter__` method,

... and with a `__next__` method

Iterators and Generators -1

Iterators and **generators** are tools to generate sequences of objects in an efficient way.

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
```

An `iterator` is a class ...

... provided with an `__iter__` method,

... and with a `__next__` method

```
In [1]: from fibo1 import Fibo

In [2]: 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


In [3]: fb = list(Fibo(900))

In [4]: fb
Out[4]: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]
```

Iterators and Generators -1

Iterators and **generators** are tools to generate sequences of objects in an efficient way.

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
```

An `iterator` is a class ...

... provided with an `__iter__` method,

... and with a `__next__` method

```
In [1]: from fibo1 import Fibo

In [2]: 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


In [3]: fb = list(Fibo(900))

In [4]: fb
Out[4]: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]
```

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

Iterators and Generators -1

Iterators and **generators** are tools to generate sequences of objects in an efficient way.

file: fibo1.py

An `iterator` is a class ...

... provided with an `__iter__` method,

... and with a `__next__` method

```
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
```

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

You can also get a list from an `iterator`

```
In [1]: from fibo1 import Fibo

In [2]: 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


In [3]: fb = list(Fibo(900))

In [4]: fb
Out[4]: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]
```

Iterators and Generators -1

Iterators and **generators** are tools to generate sequences of objects in an efficient way.

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
```

An `iterator` is a class ...

... provided with an `__iter__` method,

... and with a `__next__` method

```
In [1]: from fibo1 import Fibo

In [2]: 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


In [3]: fb = list(Fibo(900))

In [4]: fb
Out[4]: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]
```

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

You can also get a list from an `iterator`

⟶

Iterators and Generators -2

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
```

Iterators and Generators -2

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

file: fibo2.py

A `generator` looks like a function ...

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

Iterators and Generators -2

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

file: fibo2.py

A `generator` looks like a function ...

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

... returning values with the **yield** statement

Iterators and Generators -2

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

file: fibo2.py

A `generator` looks like a function ...

... returning values with the **yield** statement

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

```
In [1]: from fibo2 import fibo

In [2]: 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
```

Iterators and Generators -2

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

file: fibo2.py

A `generator` looks like a function ...

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

... returning values with the **yield** statement

```
In [1]: from fibo2 import fibo

In [2]: 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
```

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

Iterators and Generators -2

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

file: fibo2.py

A `generator` looks like a function ...

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

... returning values with the **yield** statement

```
In [1]: from fibo2 import fibo

In [2]: 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
```

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

comprehensions are another form of simple iterators

Iterators and Generators -2

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

file: fibo2.py

A `generator` looks like a function ...

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

... returning values with the **yield** statement

```
In [1]: from fibo2 import fibo

In [2]: 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
```

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

comprehensions are another form of simple iterators

A `file` object is an `iterator`:

```
In [2]: fpt = open("selected.dat")

In [3]: for line in fpt:
   ...:     print(line)
   ...:
```

Iterators and Generators -2

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

file: fibo2.py

A `generator` looks like a function ...

... returning values with the **yield** statement

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

```
In [1]: from fibo2 import fibo

In [2]: 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
```

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

comprehensions are another form of simple iterators

A `file` object is an `iterator`:

```
In [2]: fpt = open("selected.dat")

In [3]: for line in fpt:
   ...:     print(line)
   ...:
```

→

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.

File access example:

```
In [2]: fpt = open("selected.dat")

In [3]: for line in fpt:
   ...:     print(line)
   ...:
In [4]: 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.

File access example:

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

```
In [2]: fpt = open("selected.dat")

In [3]: for line in fpt:
   ...:     print(line)
   ...:
In [4]: 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.

File access example:

```
In [2]: fpt = open("selected.dat")

In [3]: for line in fpt:
   ...:     print(line)
   ...:
In [4]: fpt.close()
```

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

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.

File access example:

```
In [2]: fpt = open("selected.dat")

In [3]: for line in fpt:
   ...:     print(line)
   ...:
In [4]: fpt.close()
```

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

And then **close** it, when you're done

A context manager provides for automatic release of resources.
Because a `file` object is also a content 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.

File access example:

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

```
In [2]: fpt = open("selected.dat")

In [3]: for line in fpt:
   ...:     print(line)
   ...:
In [4]: fpt.close()
```

And then **close** it, when you’re done

A context manager provides for automatic release of resources.

Because a `file` object is also a content manager:

```
In [5]: with open("selected.dat") as fpt:
   ...:     for line in fpt:
   ...:         print(line)
   ...:
```

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.

File access example:

```
In [2]: fpt = open("selected.dat")

In [3]: for line in fpt:
   ...:     print(line)
   ...:
In [4]: fpt.close()
```

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

And then **close** it, when you're done

A context manager provides for automatic release of resources.
Because a `file` object is also a content manager:

```
In [5]: with open("selected.dat") as fpt:
   ...:     for line in fpt:
   ...:         print(line)
   ...:
```

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.

File access example:

```
In [2]: fpt = open("selected.dat")

In [3]: for line in fpt:
   ...:     print(line)
   ...:
In [4]: fpt.close()
```

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

And then **close** it, when you're done

A context manager provides for automatic release of resources.
Because a `file` object is also a content manager:

```
In [5]: with open("selected.dat") as fpt:
   ...:     for line in fpt:
   ...:         print(line)
   ...:
```

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

You need not to bother closing the file, because the context manager does it for you: <u>even in case of errors!</u>

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

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

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

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

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

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

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

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

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

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

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

... 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

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

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`

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

```
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

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 t
    return wrapper
```

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

... and returning a function

Decorators

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 t
    return wrapper
```

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

The returned function does “something” with the original one

... and returning a function

Decorators

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 t
    return wrapper
```

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

In this case:
- gets the time
- calls the original function
- computes and prints the elapsed time

... and returning a function

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

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 t
    return wrapper
```

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

In this case:
- gets the time
- calls the original function
- computes and prints the elapsed time

... and returning a function

Example: decorating a function - File: fibo3.py

```
from decorato

@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()
```

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

Decorators

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 t
    return wrapper
```

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

In this case:
- gets the time
- calls the original function
- computes and prints the elapsed time

... and returning a function

Example: decorating a function - File: fibo3.py

```
from decorato

@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()
```

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

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

Standard Modules and Packages

- Python is particularly rich in modules and packages distributed together with the language interpreter.

Standard Modules and Packages

- Python is particularly rich in modules and packages distributed together with the language interpreter.
- The first thing to do when starting a new program is: look for a module which solves your problem

Standard Modules and Packages

- Python is particularly rich in modules and packages distributed together with the language interpreter.
- The first thing to do when starting a new program is: look for a module which solves your problem
- Standard modules: (⇒)
  - sys
  - os
  - os.path
  - math, cmath
  - random
  - ... plus 280, more or less

Standard Modules and Packages

- Python is particularly rich in modules and packages distributed together with the language interpreter.
- The first thing to do when starting a new program is: look for a module which solves your problem
- Standard modules: (⇒)
  - sys
  - os
  - os.path
  - math, cmath
  - random
  - ... plus 280, more or less
- Interesting packages:
  - numpy
  - scipy
  - matplotlib
  - astropy

Standard Modules and Packages

- Python is particularly rich in modules and packages distributed together with the language interpreter.
- The first thing to do when starting a new program is: look for a module which solves your problem
- Standard modules: (⇒)
  - sys
  - os
  - os.path
  - math, cmath
  - random
  - ... plus 280, more or less
- Interesting packages:
  - numpy
  - scipy
  - matplotlib
  - astropy

The **sys** standard module

Classes, functions and constants directly related with the `Python` interpreter.

Let's explore at the `python` prompt:

- sys.path (See also: PYTHONPATH)
- sys.argv
- sys.exit()

- sys.maxint
- sys.float_info
- sys.maxsize
- sys.platform

- sys.stdin
- sys.stdout
- sys.stderr

The os standard package

Classes, functions and constants directly related with the Operating System.

Let’s explore at the `python` prompt:

- os.sep
- os.linesep
- os.defpath
- os.environ
- os.getenv(”HOME”)
- os.tmpfile()
- os.access(”a.py”,os.R_OK)
- os.curdir
- os.chdir(”newdir”)
- os.listdir(”dirname”)
- os.mkdir(”/dir1/dir2/name”)
- os.makedirs(”/dir1/dir2/name”)
- os.rename(”old”,”new”)
- os.renames(”old”,”new”)
- os.walk(”topdir”)

The os standard package

Classes, functions and constants directly related with the Operating System.

Let’s explore at the `python` prompt:

- os.sep
- os.linesep
- os.defpath
- os.environ
- os.getenv(”HOME”)
- os.tmpfile()
- os.access(”a.py”,os.R_OK)
- os.curdir
- os.chdir(”newdir”)
- os.listdir(”dirname”)
- os.mkdir(”/dir1/dir2/name”)
- os.maked
- os.renam
- os.renam
- os.walk(”topdir”)

```
import os

tree=os.walk(".")
for dp,dnames,fnames in tree:
    for fn in fnames:
       print(os.path.join(dp,fn))
```

The `os.path` standard module

Function to manipulate file names and paths in a portable way

Let's explore at the `python` prompt:

- os.path.abspath(path)
- os.path.basename(path)
- os.path.dirname(path)
- os.path.split(path)
- os.path.commonprefix(list)
- os.path.exists(path)
- os.path.getatime(path)
- os.path.getmtime(path)
- os.path.getctime(path)
- os.path.join(path1,path2,..)

Modules: `math`, `cmath`, `random`

Functions on real numbers (`math`) complex numbers (`cmath`) and random numbers generation (`random`)

Let's explore at the `python` prompt:

- help(math), to be noted:
  - fsum
  - expm1
  - log1p
- help(cmath)
- help(random)

Exceptions -1

Errors in python programs are managed by means of exceptions.

file: error.py

```
def division(a, b):
    return a/b
```

Exceptions -1

Errors in python programs are managed by means of exceptions.

file: error.py

```
def division(a, b):
    return a/b
```

Now let’s force an error:

```
>>> from error import division
>>> division(2.0, 4)
0.5
>>> division(2.0, 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "error.py", line 2, in division
    return a/b
ZeroDivisionError: float division by zero
```

Exceptions -1

Errors in python programs are managed by means of exceptions.

file: error.py

```
def division(a, b):
    return a/b
```

Now let's force an error:

```
>>> from error import division
>>> division(2.0, 4)
0.5
>>> division(2.0, 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "error.py", line 2, in division
    return a/b
ZeroDivisionError: float division by zero
```

Exceptions can be catch:

file: error1.py

```
import error

def division(a,b):
    try:
        return error.division(a, b)
    except:
        print("You can't divide by zero!")
```

Exceptions -1

Errors in python programs are managed by means of exceptions.

file: error.py

```
def division(a, b):
    return a/b
```

Now let's force an error:

```
>>> from error import division
>>> division(2.0, 4)
0.5
>>> division(2.0, 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "error.py", line 2, in division
    return a/b
ZeroDivisionError: float division by zero
```

Exceptions can be catch:

file: error1.py

```
import error

def division(a,b):
    try:
        return error.division(a, b)
    except:
        print("You can't divide by zero!")
```

And here it is:

```
>>> from error1 import division
>>> division(2.33, 0)
You can't divide by zero!
>>>
```

Exceptions -1

Errors in python programs are managed by means of exceptions.

file: error.py

```
def division(a, b):
    return a/b
```

Now let’s force an error:

```
>>> from error import division
>>> division(2.0, 4)
0.5
>>> division(2.0, 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "error.py", line 2, in division
    return a/b
ZeroDivisionError: float division by zero
```

Exceptions can be catch:

file: error1.py

```
import error

def division(a,b):
    try:
        return error.division(a, b)
    except:
        print("You can’t divide by zero!")
```

And here it is:

```
>>> from error1 import division
>>> division(2.33, 0)
You can’t divide by zero!
>>>
```

Exceptions -1

Errors in python programs are managed by means of exceptions.

file: error.py

```
def division(a, b):
    return a/b
```

Now let’s force an error:

```
>>> from error import division
>>> division(2.0, 4)
0.5
>>> division(2.0, 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "error.py", line 2, in division
    return a/b
ZeroDivisionError: float division by zero
```

Exceptions can be catch:

file: error1.py

```
import error

def division(a,b):
    try:
        return error.div
    except:
        print("You can’t
```

Note that the exception mechanism allows the programmer to manage the error at the proper level.

In the above example the exception is catch in the caller of the function where the error happens.

The principle is: in case of error the exception climbs up the sequence of nested calls until catch. If it is not catch somewhere, the program terminates with proper error message.

And here it is:

```
>>> from error1 import
>>> division(2.33, 0)
You can’t divide by zero!
>>>
```

Exceptions -2

Let’s try something different:

```
>>> division("two", 2)
You can’t divide by zero!
>>>
```

Exceptions -2

Let’s try something di

Our error management is not proper when errors other than zero division happen

```
>>> division("two", 2)
You can’t divide by zero!
>>>
```

Exceptions -2

Let’s try something di

```
>>> division("two", 2)
You can’t divide by zero!
>>>
```

Our error management is not proper when errors other than zero division happen

The exception mechanism provides for more detailed management:

file: error2.py

```python
import error

def division(a,b):
    try:
        return error.division(a, b)
    except ZeroDivisionError:
        print("You can’t divide by zero!")
```

Exceptions -2

Let's try something di

```
>>> division("two", 2)
You can't divide by zero!
>>>
```

Our error management is not proper when errors other than zero division happen

The exception mechanism provides for more detailed management:

file: error2.py

```
import error

def division(a,b):
    try:
        return error.division(a,
    except ZeroDivisionError:
        print("You can't divide by zero!")
```

Exceptions have a type. Here we only catch the exception `ZeroDivisionError`.

Exceptions -2

Let’s try something di

```
>>> division("two", 2)
You can’t divide by zero!
>>>
```

Our error management is not proper when errors other than zero division happen

The exception mechanism provides for more detailed management:

file: error2.py

```
import error

def division(a,b):
    try:
        return error.division(a,
    except ZeroDivisionError:
        print("You can’t divide by zero!")
```

Exceptions have a type. Here we only catch the exception `ZeroDivisionError`.

Now everything works better:

```
>>> from error2 import division
>>> division(2, 0)
You can’t divide by zero!
>>> division("two", 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/lfini/Personale/CorsiSeminari/2018-Python/varie/error2.py", line 5
    error.division(a, b)
  File "/home/lfini/Personale/CorsiSeminari/2018-Python/varie/error.py", line 2,
    return a/b
TypeError: unsupported operand type(s) for /: ’str’ and ’int’
```

Exceptions -2

Let's try something di

```
>>> division("two", 2)
You can't divide by zero!
>>>
```

Our error management is not proper when errors other than zero division happen

The exception mechanism provides for more detailed management:

file: error2.py

```
import error

def division(a,b):
    try:
        return error.division(a,
    except ZeroDivisionError:
        print("You can't divide by zero!")
```

Exceptions have a type. Here we only catch the exception `ZeroDivisionError`.

Now everything works better:

```
>>> from error2 import divi
>>> division(2, 0)
You can't divide by zero!
>>> division("two", 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/lfini/Personale/CorsiSeminari/2018-Python/varie/error2.py", line 5
    error.division(a, b)
  File "/home/lfini/Personale/CorsiSeminari/2018-Python/varie/error.py", line 2,
    return a/b
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

⟸ The exception `ZeroDivisionError` is catch and managed by the program.

Exceptions -2

Let’s try something di

```
>>> division("two", 2)
You can’t divide by zero!
>>>
```

Our error management is not proper when errors other than zero division happen

The exception mechanism provides for more detailed management:

file: error2.py

```
import error

def division(a,b):
    try:
        return error.division(a,
    except ZeroDivisionError:
        print("You can’t divide by zero!")
```

Exceptions have a type. Here we only catch the exception `ZeroDivisionError`.

Now everything

```
>>> from error2 import divi
>>> division(2, 0)
You can’t divide by zero!
>>> division("two", 2)
Traceback (most recent cal
  File "<stdin>", line 1,
  File "/home/lfini/Persona                                   .py", line 5
    error.division(a, b)
  File "/home/lfini/Personale/CorsiSeminari/2018-Python/varie/error.py", line 2,
    return a/b
TypeError: unsupported operand type(s) for /: ’str’ and ’int’
```

The exception `ZeroDivisionError` is catch and managed by the program.

Any other exception is not managed, and causes program termination.

Exceptions -2

Let's try something di

```
>>> division("two", 2)
You can't divide by zero!
>>>
```

Our error management is not proper when errors other than zero division happen

The exception mechanism provides for more detailed management:

file: error2.py

```
import error

def division(a,b):
    try:
        return error.division(a,
    except ZeroDivisionError:
        print("You can't divide by zero!")
```

Exceptions have a type. Here we only catch the exception `ZeroDivisionError`.

Now everything works better:

```
>>> from error2 import divi
>>> division(2, 0)
You can't divide by zero!
>>> division("two", 2)
Traceback (most recent cal
  File "<stdin>", line 1, i
  File "/home/lfini/Persona                                              .py", line 5
    error.division(a, b)
  File "/home/lfini/Personale/CorsiSeminari/2018-Python/varie/error.py", line 2,
    return a/b
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

The exception `ZeroDivisionError` is catch and managed by the program.

Any other exception is not managed, and causes program termination.

→

Debug and test

Python is provided with a debugger (**pdb**) which allows to control program execution:

- defining breakpoints
- executing a program step by step
- looking into variables
- executing functions at the prompt

Let's see an example:

```
 pdb fibo.py
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(1)<module>()
-> "Module for the computation of Fibonacci series"
(Pdb) b 9
Breakpoint 1 at /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py:9
(Pdb) c
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(9)fibo()
-> a, b = 0, 1
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(10)fibo()
-> while b < n:
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(11)fibo()
-> result.append(b)
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(12)fibo()
-> a, b = b, a+b
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(10)fibo()
-> while b < n:
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(11)fibo()
-> result.append(b)
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(12)fibo()
-> a, b = b, a+b
(Pdb) p result
[1, 1]
```

Debug and test

Python is provided with a debugger (**pdb**) which allows to control program execution:

- defining breakpoints
- executing a program step by step
- looking into variables
- executing functions at the prompt

Let's see an example:

```
 pdb fibo.py
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(1)<module>()
-> "Module for the computation of Fibonacci series"
(Pdb) b 9
Breakpoint 1 at /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py:9
(Pdb) c
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(9)fibo()
-> a, b = 0, 1
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(10)fibo()
-> while b < n:
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(11)fibo()
-> result.append(b)
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(12)fibo()
-> a, b = b, a+b
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(10)fibo()
-> while b < n:
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(11)fibo()
-> result.append(b)
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(12)fibo()
-> a, b = b, a+b
(Pdb) p result
[1, 1]
```

Debug and test

Python is provided with a debugger (**pdb**) which allows to control program execution:

- defining breakpoints
- executing a program step by step
- looking into variables
- executing functions at the prompt

Let's see an e

Start the debugger on program `fibo.py`. The debugger stops immediately at the first line of the program

```
 pdb fibo.py
> /home/lfini/Perso                                                         dule>()
-> "Module for the computation of Fibonacci series"
(Pdb) b 9
Breakpoint 1 at /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py:9
(Pdb) c
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(9)fibo()
-> a, b = 0, 1
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(10)fibo()
-> while b < n:
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(11)fibo()
-> result.append(b)
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(12)fibo()
-> a, b = b, a+b
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(10)fibo()
-> while b < n:
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(11)fibo()
-> result.append(b)
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(12)fibo()
-> a, b = b, a+b
(Pdb) p result
[1, 1]
```

Debug and test

Python is provided with a debugger (**pdb**) which allows to control program execution:

- defining breakpoints
- executing a program step by step
- looking into variables
- executing functions at the prompt

Let’s see an e

Start the debugger on program `fibo.py`. The debugger stops immediately at the first line of the program

Set a breakpoint at line number 9

```
 pdb fibo.py
> /home/lfini/Perso                                               dule>()
-> "Module for
(Pdb) b 9
Breakpoint 1 at /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py:9
(Pdb) c
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(9)fibo()
-> a, b = 0, 1
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(10)fibo()
-> while b < n:
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(11)fibo()
-> result.append(b)
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(12)fibo()
-> a, b = b, a+b
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(10)fibo()
-> while b < n:
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(11)fibo()
-> result.append(b)
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(12)fibo()
-> a, b = b, a+b
(Pdb) p result
[1, 1]
```

Debug and test

Python is provided with a debugger (**pdb**) which allows to control program execution:

- defining breakpoints
- executing a program step by step
- looking into variables
- executing functions at the prompt

Let's see an e

Start the debugger on program `fibo.py`. The debugger stops immediately at the first line of the program

Set a breakpoint at line number 9

Continue (actually: start) the execution

```
 pdb fibo.py
> /home/lfini/Perso                                                 dule>()
-> "Module for 
(Pdb) b 9
Breakpoint 1 at                                              on/varie/fibo.py:9
(Pdb) c
> /home/lfini/                                                 py(9)fibo()
-> a, b = 0, 1
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(10)fibo()
-> while b < n:
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(11)fibo()
-> result.append(b)
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(12)fibo()
-> a, b = b, a+b
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(10)fibo()
-> while b < n:
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(11)fibo()
-> result.append(b)
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(12)fibo()
-> a, b = b, a+b
(Pdb) p result
[1, 1]
```

Debug and test

Python is provided with a debugger (**pdb**) which allows to control program execution:

- defining breakpoints
- executing a program step by step
- looking into variables
- executing functions at the prompt

Let’s see an e

Start the debugger on program `fibo.py`. The debugger stops immediately at the first line of the program

Set a breakpoint at line number 9

Con

The debugger stops at the breakpoint (just before executing line 9)

```
 pdb fibo.py
> /home/lfini/Perso                                            dule>()
-> "Module for
(Pdb) b 9
Breakpoint 1 at                                        ython/varie/fibo.py:9
(Pdb) c
> /home/lfini/                                                 bo()
-> a, b = 0, 1
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(10)fibo()
-> while b < n:
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(11)fibo()
-> result.append(b)
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(12)fibo()
-> a, b = b, a+b
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(10)fibo()
-> while b < n:
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(11)fibo()
-> result.append(b)
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(12)fibo()
-> a, b = b, a+b
(Pdb) p result
[1, 1]
```

Debug and test

Python is provided with a debugger (**pdb**) which allows to control program execution:

- defining breakpoints
- executing a program step by step
- looking into variables
- executing functions at the prompt

Let's see an e

Start the debugger on program `fibo.py`. The debugger stops immediately at the first line of the program

Set a breakpoint at line number 9

Con

The debugger stops at the breakpoint (just before executing line 9)

Execute a number of steps

```
 pdb fibo.py
> /home/lfini/Perso                                        dule>()
-> "Module for
(Pdb) b 9
Breakpoint 1 at                                         Python/varie/fibo.py:9
(Pdb) c
> /home/lfini/                                          bo()
-> a, b = 0, 1
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(10)fibo()
-> while b <
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(11)fibo()
-> result.append(b)
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(12)fibo()
-> a, b = b, a+b
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(10)fibo()
-> while b < n:
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(11)fibo()
-> result.append(b)
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(12)fibo()
-> a, b = b, a+b
(Pdb) p result
[1, 1]
```

Debug and test

Python is provided with a debugger (**pdb**) which allows to control program execution:

- defining breakpoints
- executing a program step by step
- looking into variables
- executing functions at the prompt

Let’s see an e

Start the debugger on program `fibo.py`. The debugger stops immediately at the first line of the program

Set a breakpoint at line number 9

Con

The debugger stops at the breakpoint (just before executing line 9)

Execute a number of steps

```
 pdb fibo.py
> /home/lfini/Perso                                                    dule>()
-> "Module for
(Pdb) b 9
Breakpoint 1 at                                          Python/varie/fibo.py:9
(Pdb) c
> /home/lfini/                                                          bo()
-> a, b = 0, 1
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(10)fibo()
-> while b <
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(11)fibo()
-> result.append(b)
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(12)fibo()
-> a, b = b, a+b
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(10)fibo()
-> while b < n:
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(11)fibo()
-> result.append(b)
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(12)fibo()
-> a, b = b, a+b
(Pdb) p result
[1, 1]
```

Debug and test

Python is provided with a debugger (**pdb**) which allows to control program execution:

- defining breakpoints
- executing a program step by step
- looking into variables
- executing functions at the prompt

Let's see an e

Start the debugger on program `fibo.py`. The debugger stops immediately at the first line of the program

Set a breakpoint at line number 9

Con

The debugger stops at the breakpoint (just before executing line 9)

Execute a number of steps

```
 pdb fibo.py
> /home/lfini/Perso                                                dule>()
-> "Module for
(Pdb) b 9
Breakpoint 1 at                                           Python/varie/fibo.py:9
(Pdb) c
> /home/lfini/                                                 bo()
-> a, b = 0, 1
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(10)fibo()
-> while b <
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(11)fibo()
-> result.append(b)
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(12)fibo()
-> a, b = b, a+b
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(10)fibo()
-> while b < n:
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(11)fibo()
-> result.append(b)
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(12)fibo()
-> a, b = b, a+b
(Pdb) p result
[1, 1]
```

Debug and test

Python is provided with a debugger (**pdb**) which allows to control program execution:

- defining breakpoints
- executing a program step by step
- looking into variables
- executing functions at the prompt

Let's see an e

Start the debugger on program `fibo.py`. The debugger stops immediately at the first line of the program

Set a breakpoint at line number 9

Con

The debugger stops at the breakpoint (just before executing line 9)

Execute a number of steps

```
 pdb fibo.py
> /home/lfini/Perso                                              dule>()
-> "Module for
(Pdb) b 9
Breakpoint 1 at                                               ython/varie/fibo.py:9
(Pdb) c
> /home/lfini/                                                 bo()
-> a, b = 0, 1
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(10)fibo()
-> while b <
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(11)fibo()
-> result.append(b)
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(12)fibo()
-> a, b = b, a+b
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(10)fibo()
-> while b < n:
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(11)fibo()
-> result.append(b)
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(12)fibo()
-> a, b = b, a+b
(Pdb) p result
[1, 1]
```

Debug and test

Python is provided with a debugger (**pdb**) which allows to control program execution:

- defining breakpoints
- executing a program step by step
- looking into variables
- executing functions at the prompt

Let's see an e

Start the debugger on program `fibo.py`. The debugger stops immediately at the first line of the program

Set a breakpoint at line number 9

Con

The debugger stops at the breakpoint (just before executing line 9)

Execute a number of steps

```
 pdb fibo.py
> /home/lfini/Perso                                               dule>()
-> "Module for
(Pdb) b 9
Breakpoint 1 at                                                  Python/varie/fibo.py:9
(Pdb) c
> /home/lfini/                                                   bo()
-> a, b = 0, 1
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(10)fibo()
-> while b <
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(11)fibo()
-> result.append(b)
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(12)fibo()
-> a, b = b, a+b
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(10)fibo()
-> while b < n:
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(11)fibo()
-> result.append(b)
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(12)fibo()
-> a, b = b, a+b
(Pdb) p result
[1, 1]
```

Debug and test

Python is provided with a debugger (**pdb**) which allows to control program execution:

- defining breakpoints
- executing a program step by step
- looking into variables
- executing functions at the prompt

Let’s see an e

```
 pdb fibo.py
> /home/lfini/Person                                        dule>()
-> "Module for
(Pdb) b 9
Breakpoint 1 at                                      Python/varie/fibo.py:9
(Pdb) c
> /home/lfini/                                                  bo()
-> a, b = 0, 1
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(10)fibo()
-> while b <
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(11)fibo()
-> result.append(b)
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(12)fibo()
-> a, b = b, a+b
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(10)fibo()
-> while b < n:
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(11)fibo()
-> result.append(b)
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(12)fibo()
-> a, b = b, a+b
(Pdb) p result
[1, 1]
```

Start the debugger on program `fibo.py`. The debugger stops immediately at the first line of the program

Set a breakpoint at line number 9

Con

The debugger stops at the breakpoint (just before executing line 9)

Execute a number of steps

Debug and test

Python is provided with a debugger (**pdb**) which allows to control program execution:

- defining breakpoints
- executing a program step by step
- looking into variables
- executing functions at the prompt

Let's see an e

Start the debugger on program `fibo.py`. The debugger stops immediately at the first line of the program

```
 pdb fibo.py
> /home/lfini/Persor                                          dule>()
-> "Module for
```

Set a breakpoint at line number 9

```
(Pdb) b 9
Breakpoint 1 at                                           Python/varie/fibo.py:9
(Pdb) c
```

Con

The debugger stops at the breakpoint (just before executing line 9)

```
> /home/lfini/                                                bo()
-> a, b = 0, 1
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(10)fibo()
-> while b <
```

Execute a number of steps

```
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(11)fibo()
-> result.append(b)
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(12)fibo()
-> a, b = b, a+b
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(10)fibo()
-> while b < n:
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(11)fibo()
-> result.append(b)
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2018-Python/varie/fibo.py(12)fibo()
-> a, b = b, a+b
(Pdb) p result
[1, 1]
```

Now see what's in variable `result`