Introduction to Python - I

Overview

- General aspects
- Getting and installing Python
- Elementary use of Python
- Expressions and statements
- Collections
- Functions and arguments

Overview

- General aspects
- Getting and installing Python
- Elementary use of Python
- Expressions and statements
- Collections
- Functions and arguments

→

Python generalities -1

- A scripting language

Python generalities -1

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

Python generalities -1

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

Python generalities -1

- A scripting language
  - ... but not only
    - Numerical computations (!!)
    - CGI Applications
    - GUI programming
    - Gaming
    - Automation
    - Artificial intelligence
- Supports different programming models
  - Procedural
  - Object Oriented
  - Functional (... somewhat)
- Semi-interpreted language
  - Automatic p-code management
  - p-code files generated in sub-directory `__pycache__` (since Python 3.7, p-code location is configurable)

Python generalities -1

- A scripting language
  - ... but not only
    - Numerical computations (!!)
    - CGI Applications
    - GUI programming
    - Gaming
    - Automation
    - Artificial intelligence
- Supports different programming models
  - Procedural
  - Object Oriented
  - Functional (... somewhat)
- Semi-interpreted language
  - Automatic p-code management
  - p-code files generated in sub-directory `__pycache__` (since Python 3.7, p-code location is configurable)

→

Python generalities -2

- "Object Oriented" language
  - Full support of OO programming
  - "Everything is an object"

Python generalities -2

- “Object Oriented” language
  - Full support of OO programming
  - “Everything is an object”
- “Typed” language
  - Objects have a type
  - Identifiers (names) are not typed

Python generalities -2

- “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”

Python generalities -2

- "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"
- Huge standard library
  - Well portable
  - Covers the largest areas of applications

Python generalities -2

- “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”
- Huge standard library
  - Well portable
  - Covers the largest areas of applications
- Allows interactive use
  - Using the interpreter
  - Using `ipython`

Python generalities -2

- “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”
- Huge standard library
  - Well portable
  - Covers the largest areas of applications
- 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
- C#
- 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
- C#
- Perl
- Ruby
- Php
- R
- Matlab ($)
- Octave
- awk
- IDL ($)
- ....

→

Versions and Platforms

- Versions
  - **3.9.0** - The “latest stable” version[1]
  - **2.7.x** - Still sometimes used[2]. Now frozen[3]

Versions and Platforms

- Versions
  - **3.9.0** - The “latest stable” version[1]
  - **2.7.x** - Still sometimes used[2]. Now frozen[3]
- Platforms

Versions and Platforms

- Versions
  - **3.9.0** - The "latest stable" version[1]
  - **2.7.x** - Still sometimes used[2]. Now frozen[3]
- Platforms
  - Linux (3.8.x preinstalled)

Versions and Platforms

- Versions
  - **3.9.0** - The "latest stable" version[1]
  - **2.7.x** - Still sometimes used[2]. Now frozen[3]
- Platforms
  - Linux (3.8.x preinstalled)
  - Windows and .NET
    - Installer at `www.python.org`
    - Active State (Commercial)
    - IronPython

Versions and Platforms

- Versions
  - **3.9.0** - The “latest stable” version[1]
  - **2.7.x** - Still sometimes used[2]. Now frozen[3]
- Platforms
  - Linux (3.8.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”

Versions and Platforms

- Versions
  - **3.9.0** - The "latest stable" version[1]
  - **2.7.x** - Still sometimes used[2]. Now frozen[3]
- Platforms
  - Linux (3.8.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[4]
    - buildozer[3]
    - SLA4

Versions and Platforms

- Versions
  - **3.9.0** - The “latest stable” version[1]
  - **2.7.x** - Still sometimes used[2]. Now frozen[3]
- Platforms
  - Linux (3.8.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[4]
    - buildozer[3]
    - SLA4

[1] As of October 2020

[2] Most linux distributions have switched to Python 3 as default in 2019

[3] Since Jan 1, 2020 no updates whatsoever are provided

[4] For ”App” development

- Versions
  - **3.9.0** - The “latest stable” version[1]
  - **2.7.x** - Still sometimes used[2]. Now frozen[3]
- Platforms
  - Linux (3.8.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[4]
    - buildozer[3]
    - SLA4

[1] As of October 2020

[2] Most linux distributions have switched to Python 3 as default in 2019

[3] Since Jan 1, 2020 no updates whatsoever are provided

[4] For ”App” development

→

Python on Linux

Python on Linux

- Most Linux distributions come with Python 3.x preinstalled.

Python on Linux

- Most Linux distributions come with Python 3.x preinstalled.
- The usual way to install Python is using the software installation and upgrade utility provided in your distribution (e.g.: `apt`, `yum`, etc.).

Python on Linux

- Most Linux distributions come with Python 3.x preinstalled.
- The usual way to install Python is using the software installation and upgrade utility provided in your distribution (e.g.: `apt`, `yum`, etc.).
- Additional modules can be usually installed either with the standard installation utility (when the required module is available) or using Python's own installation utility **pip** (more about it in the following)

Python on Linux

- Most Linux distributions come with Python 3.x preinstalled.
- The usual way to install Python is using the software installation and upgrade utility provided in your distribution (e.g.: `apt`, `yum`, etc.).
- Additional modules can be usually installed either with the standard installation utility (when the required module is available) or using Python's own installation utility **pip** (more about it in the following)
- When multiple versions of python must be available on the same system, you may use **anaconda**:

  https://en.wikipedia.org/wiki/Anaconda_(Python_distribution)

Python on Linux

- Most Linux distributions come with Python 3.x preinstalled.
- The usual way to install Python is using the software installation and upgrade utility provided in your distribution (e.g.: `apt`, `yum`, etc.).
- Additional modules can be usually installed either with the standard installation utility (when the required module is available) or using Python's own installation utility **pip** (more about it in the following)
- When multiple versions of python must be available on the same system, you may use **anaconda**:

  https://en.wikipedia.org/wiki/Anaconda_(Python_distribution)

→

Python on Windows

Python on Windows

- You may find Windows installers in the download section at:

  https://www.python.org

Python on Windows

- You may find Windows installers in the download section at:

  https://www.python.org

- Both 32 bits and 64 bits versions are provided

Python on Windows

- You may find Windows installers in the download section at:

  https://www.python.org

- Both 32 bits and 64 bits versions are provided

- **pip** and **anaconda** are both available for Windows, too

Getting and installing Python -2

Python on Windows

- You may find Windows installers in the download section at:

  https://www.python.org

- Both 32 bits and 64 bits versions are provided

- **pip** and **anaconda** are both available for Windows, too

Getting and installing Python -2

Python on Windows

- You may find Windows installers in the download section at:

  https://www.python.org

- Both 32 bits and 64 bits versions are provided

- **pip** and **anaconda** are both available for Windows, too

→

Python on MacOS

Python on MacOS

- You may find a Mac OS X installers in the download section at:

  https://www.python.org

Python on MacOS

- You may find a Mac OS X installers in the download section at:

  https://www.python.org

- You may select a 64 bit version working on *maverick* and following versions

Python on MacOS

- You may find a Mac OS X installers in the download section at:

  https://www.python.org

- You may select a 64 bit version working on *maverick* and following versions
- ... or a 32 bit version compatible with *leopard* and following

Getting and installing Python -3

Python on MacOS

- You may find a Mac OS X installers in the download section at:

  https://www.python.org

- You may select a 64 bit version working on *maverick* and following versions
- ... or a 32 bit version compatible with *leopard* and following

→

Beginner's Python -1

The Python interpreter can be used interactively: Python lines are executed as soon as and end-of-line is typed

The Python interpreter can be used interactively: Python lines are executed as soon as and end-of-line is typed

```
$ python
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 2+2
4
>>> a=3
>>>
```

The Python interpreter can be used interactively: Python lines are executed as soon as and end-of-line is typed

```
$ python          ⟵ 1
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 2+2           ⟵ 2
4                 ⟵ 3
>>> a=3           ⟵ 4
>>>               ⟵ 5
```

Beginner's Python -1

The Python interpreter can be used interactively: Python lines are executed as soon as and end-of-line is typed

```
$ python          ← 1
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 2+2           ← 2
4                 ← 3
>>> a=3           ← 4
>>>               ← 5
```

1. Launching the Python interpreter

Beginner's Python -1

The Python interpreter can be used interactively: Python lines are executed as soon as and end-of-line is typed

```
$ python ⟵ 1
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 2+2 ⟵ 2
4 ⟵ 3
>>> a=3 ⟵ 4
>>> ⟵ 5
```

1. Launching the Python interpreter
2. A Python expression

Beginner's Python -1

The Python interpreter can be used interactively: Python lines are executed as soon as and end-of-line is typed

```
$ python  ← 1
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 2+2  ← 2
4  ← 3
>>> a=3  ← 4
>>>  ← 5
```

1. Launching the Python interpreter
2. A Python expression
3. Expression's result

Beginner's Python -1

The Python interpreter can be used interactively: Python lines are executed as soon as and end-of-line is typed

```
$ python ⟵ 1
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 2+2 ⟵ 2
4 ⟵ 3
>>> a=3 ⟵ 4
>>> ⟵ 5
```

1. Launching the Python interpreter
2. A Python expression
3. Expression's result
4. A Python statement

Beginner's Python -1

The Python interpreter can be used interactively: Python lines are executed as soon as and end-of-line is typed

```
$ python ⟵ 1
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 2+2 ⟵ 2
4 ⟵ 3
>>> a=3 ⟵ 4
>>> ⟵ 5
```

1. Launching the Python interpreter
2. A Python expression
3. Expression's result
4. A Python statement
5. Statements have no result

Beginner’s Python -1

The Python interpreter can be used interactively: Python lines are executed as soon as and end-of-line is typed

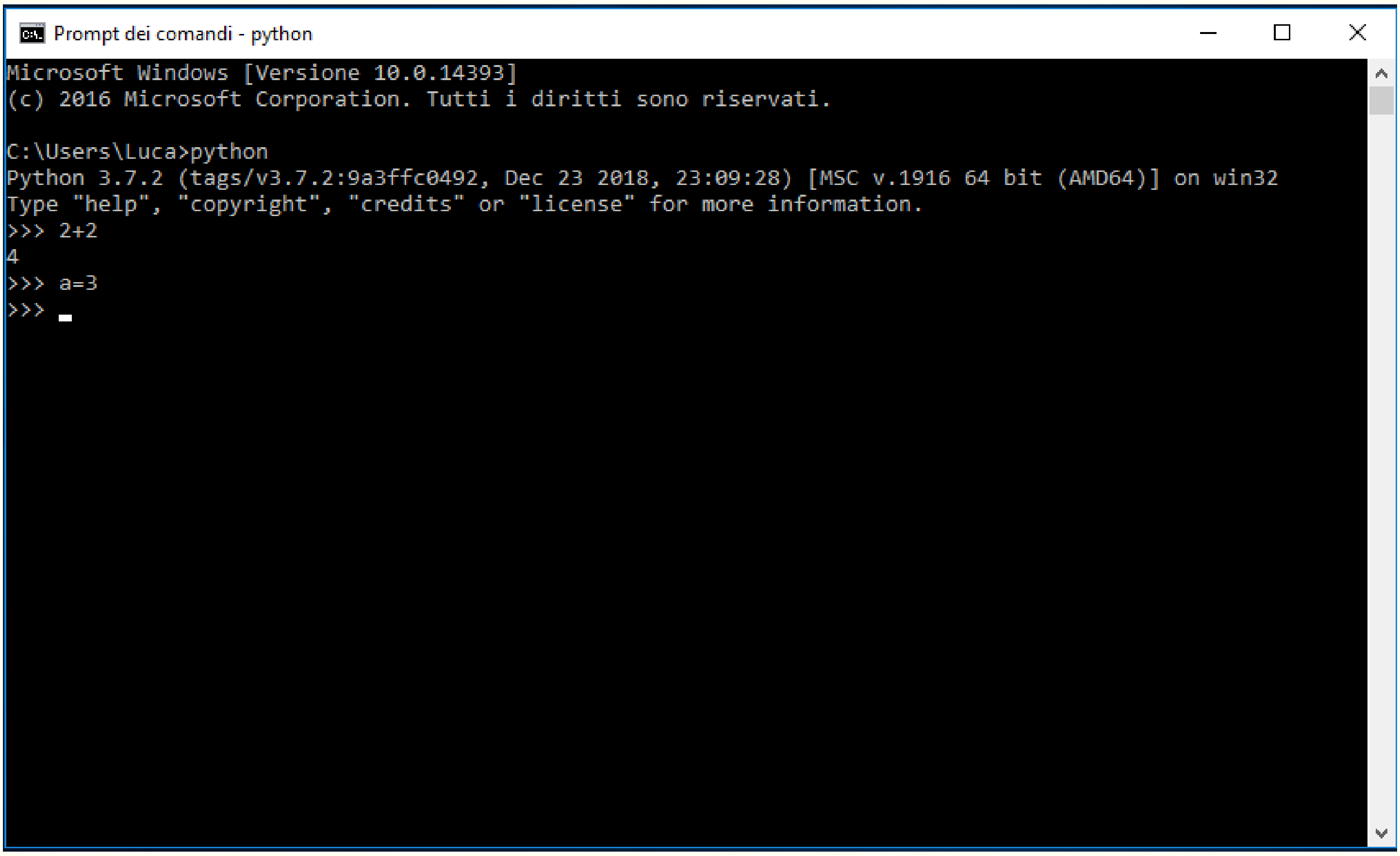
```
$ python            ← 1
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 2+2   ← 2
4  ← 3
>>> a=3   ← 4
>>>  ← 5
```

1. Launching the Python interpreter
2. A Python expression
3. Expression’s result
4. A Python statement
5. Statements have no result

→

The previous example is for Linux; the same happens on Windows:

The previous example is for Linux; the same happens on Windows:



Beginner's Python -2

The previous example is for Linux; the same happens on Windows:

Prompt dei comandi - python

```
Microsoft Windows [Versione 10.0.14393]
(c) 2016 Microsoft Corporation. Tutti i diritti sono riservati.

C:\Users\Luca>python
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 23:09:28) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 2+2
4
>>> a=3
>>>
```

→

Beginner's Python -3

Somebody may prefer to use IDLE: Python's standard integrated development system

Somebody may prefer to use IDLE: Python's standard integrated development system

```
$ idle
```

Somebody may prefer to use IDLE: Python's standard integrated development system

```
$ idle
```

IDLE shows up with Python interpreter in interactive mode, but provides many other features.
It is actually a fully usable IDE application.

Somebody may prefer to use IDLE: Python's standard integrated development system

```
$ idle
```

IDLE shows up with Python interpreter in interactive mode, but provides many other features.
It is actually a fully usable IDE application.

In the following we will be using extensively the Python interactive mode to illustrate many language features.

Somebody may prefer to use IDLE: Python's standard integrated development system

```
$ idle
```

IDLE shows up with Python interpreter in interactive mode, but provides many other features.
It is actually a fully usable IDE application.

In the following we will be using extensively the Python interactive mode to illustrate many language features.

→

Expressions and statements -1

Expressions have a value (which is written on the display in interactive mode).

Expressions have a value (which is written on the display in interactive mode).

```
>>> 3*7+5
26

>>> "Hello"+" "+"world"
'Hello world'

>>> 3 < 5
True
```

Expressions and statements -1

> Expressions have a value (which is written on the display in interactive mode).

```
>>> 3*7+5 ← 1
26

>>> "Hello"+" "+"world" ← 2
'Hello world'

>>> 3 < 5 ← 3
True
```

> Expressions have a value (which is written on the display in interactive mode).

```
>>> 3*7+5    ⟵ 1
26

>>> "Hello"+" "+"world"    ⟵ 2
'Hello world'

>>> 3 < 5    ⟵ 3
True
```

1. A numerical expression

Expressions and statements -1

Expressions have a value (which is written on the display in interactive mode).

```
>>> 3*7+5       <--  1
26

>>> "Hello"+" "+"world"   <--  2
'Hello world'

>>> 3 < 5       <--  3
True
```

1. A numerical expression
2. A string valued expression

Expressions and statements -1

Expressions have a value (which is written on the display in interactive mode).

```
>>> 3*7+5          ← 1
26

>>> "Hello"+" "+"world"   ← 2
'Hello world'

>>> 3 < 5          ← 3
True
```

1. A numerical expression
2. A string valued expression
3. A logical expression

Expressions and statements -1

Expressions have a value (which is written on the display in interactive mode).

```
>>> 3*7+5      ← 1
26

>>> "Hello"+" "+"world"      ← 2
'Hello world'

>>> 3 < 5      ← 3
True
```

1. A numerical expression
2. A string valued expression
3. A logical expression

Note: symbols may have different meanings in different expressions;

E.g.: **+** indicates either a sum in a numerical context or a concatenation of strings

Expressions and statements -1

Expressions have a value (which is written on the display in interactive mode).

```
>>> 3*7+5        ← 1
26

>>> "Hello"+" "+"world"        ← 2
'Hello world'

>>> 3 < 5        ← 3
True
```

1. A numerical expression
2. A string valued expression
3. A logical expression

Note: symbols may have different meanings in different expressions;

E.g.: **+** indicates either a sum in a numerical context or a concatenation of strings

→

Expressions and statements -2

The simplest statement is the assignment which defines a name and assigns a value to it (the result of the expression to the right of the **=** sign)

Expressions and statements -2

The simplest statement is the assignment which defines a name and assigns a value to it (the result of the expression to the right of the **=** sign)

```
>>> a = 3
>>> b = 7*a - 5 \
... + a*a/2
>>> b
20.5
```

Expressions and statements -2

The simplest statement is the assignment which defines a name and assigns a value to it (the result of the expression to the right of the **=** sign)

```
>>> a = 3  ← 1
>>> b = 7*a - 5 \  ← 2
... + a*a/2  ← 3
>>> b
20.5
```

Expressions and statements -2

The simplest statement is the assignment which defines a name and assigns a value to it (the result of the expression to the right of the **=** sign)

```
>>> a = 3            ← 1
>>> b = 7*a - 5 \    ← 2
... + a*a/2          ← 3
>>> b
20.5
```

1. Simple assignment statement

The simplest statement is the assignment which defines a name and assigns a value to it (the result of the expression to the right of the **=** sign)

```
>>> a = 3          <-- 1
>>> b = 7*a - 5 \  <-- 2
... + a*a/2        <-- 3
>>> b
20.5
```

1. Simple assignment statement
2. Python statements terminate at the end of the line

Expressions and statements -2

The simplest statement is the assignment which defines a name and assigns a value to it (the result of the expression to the right of the **=** sign)

```
>>> a = 3            ← 1
>>> b = 7*a - 5 \    ← 2
... + a*a/2          ← 3
>>> b
20.5
```

1. Simple assignment statement
2. Python statements terminate at the end of the line
3. But they may be continued by putting a \ sign at the end

Expressions and statements -2

The simplest statement is the assignment which defines a name and assigns a value to it (the result of the expression to the right of the = sign)

```
>>> a = 3           ← 1
>>> b = 7*a - 5 \   ← 2
... + a*a/2         ← 3
>>> b
20.5
```

1. Simple assignment statement
2. Python statements terminate at the end of the line
3. But they may be continued by putting a \ sign at the end

Statements do not provide a value

Expressions and statements -2

The simplest statement is the assignment which defines a name and assigns a value to it (the result of the expression to the right of the **=** sign)

```
>>> a = 3
>>> b = 7*a - 5 \
... + a*a/2
>>> b
20.5
```

1. Simple assignment statement
2. Python statements terminate at the end of the line
3. But they may be continued by putting a \ sign at the end

Statements do not provide a value

Let’s look deeper into assignment

```
>>> a = 2+1
>>> a = "three"
>>> p = [4,2,3,1]
>>> k = p
```

Expressions and statements -3

Let’s look deeper into assignment

```
>>> a = 2+1          ← 1
>>> a = "three"      ← 2
>>> p = [4,2,3,1]    ← 3
>>> k = p            ← 4
```

Let's look deeper into assignment

```
>>> a = 2+1         <- 1
>>> a = "three"     <- 2
>>> p = [4,2,3,1]   <- 3
>>> k = p           <- 4
```

1. Creates an object: "integer number valued 3" and gives it the name **a**

Let's look deeper into assignment

```
>>> a = 2+1          ← 1
>>> a = "three"      ← 2
>>> p = [4,2,3,1]    ← 3
>>> k = p            ← 4
```

1. Creates an object: "integer number valued 3" and gives it the name **a**
2. Creates an object: "character string valued 'three'" and gives it the name **a**. The name **a** is reused and the object "integer number ..." above becomes unreachable

Let’s look deeper into assignment

```
>>> a = 2+1          ⟵ 1
>>> a = "three"      ⟵ 2
>>> p = [4,2,3,1]    ⟵ 3
>>> k = p            ⟵ 4
```

1. Creates an object: “integer number valued 3” and gives it the name **a**
2. Creates an object: “character string valued ’three’” and gives it the name **a**. The name **a** is reused and the object “integer number ...” above becomes unreachable
3. Creates an object: “list of four elements” and gives it the name **p**

Let's look deeper into assignment

```
>>> a = 2+1          ← 1
>>> a = "three"      ← 2
>>> p = [4,2,3,1]    ← 3
>>> k = p            ← 4
```

1. Creates an object: "integer number valued 3" and gives it the name **a**
2. Creates an object: "character string valued 'three'" and gives it the name **a**. The name **a** is reused and the object "integer number ..." above becomes unreachable
3. Creates an object: "list of four elements" and gives it the name **p**
4. Gives another name (**k**) to the same object !!

Let's look deeper into assignment

```
>>> a = 2+1           ← 1
>>> a = "three"       ← 2
>>> p = [4,2,3,1]     ← 3
>>> k = p             ← 4
```

1. Creates an object: "integer number valued 3" and gives it the name **a**
2. Creates an object: "character string valued 'three'" and gives it the name **a**. The name **a** is reused and the object "integer number ..." above becomes unreachable
3. Creates an object: "list of four elements" and gives it the name **p**
4. Gives another name (**k**) to the same object !!

> When an object becomes unreachable (i.e.: it has no associated names), it becomes candidate for "garbage collection".
>
> Garbage collection in Python is totally automatic.

Let’s look deeper into assignment

```
>>> a = 2+1          ← 1
>>> a = "three"      ← 2
>>> p = [4,2,3,1]    ← 3
>>> k = p            ← 4
```

1. Creates an object: “integer number valued 3” and gives it the name **a**
2. Creates an object: “character string valued ’three’” and gives it the name **a**. The name **a** is reused and the object “integer number ...” above becomes unreachable
3. Creates an object: “list of four elements” and gives it the name **p**
4. Gives another name (**k**) to the same object !!

> When an object becomes unreachable (i.e.: it has no associated names), it becomes candidate for “garbage collection”.
>
> Garbage collection in Python is totally automatic.

→

Proceeding with the previous example...

```
>>> k
[4, 3, 2, 1]
>>> del p[2]
>>> p
[4, 3, 1]
>>> k
[4, 3, 1]
>>> p = 0
>>> p
0
>>> k
[4, 3, 1]
```

Proceeding with the previous example...

```
>>> k
[4, 3, 2, 1]
>>> del p[2]  ← 1
>>> p
[4, 3, 1]  ← 2
>>> k
[4, 3, 1]  ← 2
>>> p = 0  ← 3
>>> p
0
>>> k
[4, 3, 1]  ← 4
```

Proceeding with the previous example...

```
>>> k
[4, 3, 2, 1]
>>> del p[2]   ← 1
>>> p
[4, 3, 1]      ← 2
>>> k
[4, 3, 1]      ← 2
>>> p = 0      ← 3
>>> p
0
>>> k
[4, 3, 1]      ← 4
```

1. The **del** statement deletes the object indicated by the argument

Proceeding with the previous example...

```
>>> k
[4, 3, 2, 1]
>>> del p[2]   ← 1
>>> p
[4, 3, 1]      ← 2
>>> k
[4, 3, 1]      ← 2
>>> p = 0      ↖
>>> p            3
0
>>> k
[4, 3, 1]      ← 4
```

1. The **del** statement deletes the object indicated by the argument
2. Both **k** and **p** are affected

Proceeding with the previous example...

```
>>> k
[4, 3, 2, 1]
>>> del p[2]  ← 1
>>> p
[4, 3, 1]  ← 2
>>> k
[4, 3, 1]  ← 2
>>> p = 0  ← 3
>>> p
0
>>> k
[4, 3, 1]  ← 4
```

1. The **del** statement deletes the object indicated by the argument
2. Both **k** and **p** are affected
3. Here the name **p** is reused

Expressions and statements -4

Proceeding with the previous example...

```
>>> k
[4, 3, 2, 1]
>>> del p[2]  ← 1
>>> p
[4, 3, 1]  ← 2
>>> k
[4, 3, 1]  ← 2
>>> p = 0  ↖ 3
>>> p
0
>>> k
[4, 3, 1]  ← 4
```

1. The **del** statement deletes the object indicated by the argument
2. Both **k** and **p** are affected
3. Here the name **p** is reused
4. **k** is (obviously) not affected

Proceeding with the previous example...

```
>>> k
[4, 3, 2, 1]
>>> del p[2]   ← 1
>>> p
[4, 3, 1]      ← 2
>>> k
[4, 3, 1]      ← 2
>>> p = 0      ↖ 3
>>> p
0
>>> k
[4, 3, 1]      ← 4
```

1. The **del** statement deletes the object indicated by the argument
2. Both **k** and **p** are affected
3. Here the name **p** is reused
4. **k** is (obviously) not affected

→

Numerical types

```
>>> a = 3                ← 1
>>> a
3
>>> b = 3.1415926        ← 2
>>> b
3.1415926
>>> float(a)             ← 3
3.0
>>> int(b)               ← 3
3
>>> c = complex(a,b)     ← 4
>>> c
(3+3.1415926j)
```

Numerical types

```
>>> a = 3            ← 1
>>> a
3
>>> b = 3.1415926    ← 2
>>> b
3.1415926
>>> float(a)         ← 3
3.0
>>> int(b)           ← 3
3
>>> c = complex(a,b) ← 4
>>> c
(3+3.1415926j)
```

1. Python provides an integer type

Numerical types

```
>>> a = 3              ← 1
>>> a
3
>>> b = 3.1415926      ← 2
>>> b
3.1415926
>>> float(a)           ← 3
3.0
>>> int(b)             ← 3
3
>>> c = complex(a,b)   ← 4
>>> c
(3+3.1415926j)
```

1. Python provides an integer type
2. A float tipe

Numerical types

```
>>> a = 3               ← 1
>>> a
3
>>> b = 3.1415926       ← 2
>>> b
3.1415926
>>> float(a)            ← 3
3.0
>>> int(b)              ← 3
3
>>> c = complex(a,b)    ← 4
>>> c
(3+3.1415926j)
```

1. Python provides an integer type
2. A float tipe
3. Conversion functions

Data types and expressions -1

Numerical types

```
>>> a = 3          ← 1
>>> a
3
>>> b = 3.1415926  ← 2
>>> b
3.1415926
>>> float(a)       ← 3
3.0
>>> int(b)         ← 3
3
>>> c = complex(a,b)  ← 4
>>> c
(3+3.1415926j)
```

1. Python provides an integer type
2. A float tipe
3. Conversion functions
4. And a complex type

Numerical types

```
>>> a = 3            ← 1
>>> a
3
>>> b = 3.1415926    ← 2
>>> b
3.1415926
>>> float(a)         ← 3
3.0
>>> int(b)           ← 3
3
>>> c = complex(a,b) ← 4
>>> c
(3+3.1415926j)
```

1. Python provides an integer type
2. A float tipe
3. Conversion functions
4. And a complex type

→

Numerical expressions

```
>>> a+b*c
(12.424777800000001+9.86960406437476j)
>>> a**2+b**2
18.869604064374762
>>> a**2-b**2+c
(2.1303959356252395+3.1415926j)
```

Data types and expressions -2

Numerical expressions

```
>>> a+b*c          ← 1
(12.424777800000001+9.86960406437476j)
>>> a**2+b**2
18.869604064374762
>>> a**2-b**2+c    ← 2
(2.1303959356252395+3.1415926j)
```

Data types and expressions -2

Numerical expressions

```
>>> a+b*c          ← 1
(12.424777800000001+9.86960406437476j)
>>> a**2+b**2
18.869604064374762
>>> a**2-b**2+c    ← 2
(2.1303959356252395+3.1415926j)
```

1. Python's numerical expressions are pretty standard

Numerical expressions

```
>>> a+b*c           ← 1
(12.424777800000001+9.86960406437476j)
>>> a**2+b**2
18.869604064374762
>>> a**2-b**2+c     ← 2
(2.1303959356252395+3.1415926j)
```

1. Python’s numerical expressions are pretty standard
2. Different (but compatible) data types are automatically converted whenever possible

Data types and expressions -2

Numerical expressions

```
>>> a+b*c       ← 1
(12.424777800000001+9.86960406437476j)
>>> a**2+b**2
18.869604064374762
>>> a**2-b**2+c       ← 2
(2.130395935625239 5+3.1415926j)
```

1. Python’s numerical expressions are pretty standard
2. Different (but compatible) data types are automatically converted whenever possible

Conditional expressions

```
>>> a = 7
>>> b = 9
>>> a if a>b else b
9
>>> c = a+(a if a>b else b)
>>> c
16
```

Data types and expressions -2

Numerical expressions

```
>>> a+b*c          ⟵ 1
(12.424777800000001+9.86960406437476j)
>>> a**2+b**2
18.869604064374762
>>> a**2-b**2+c    ⟵ 2
(2.1303959356252395+3.1415926j)
```

1. Python's numerical expressions are pretty standard
2. Different (but compatible) data types are automatically converted whenever possible

Conditional expressions

```
>>> a = 7
>>> b = 9
>>> a if a>b else b        ⟵ 1
9
>>> c = a+(a if a>b else b)  ⟵ 2
>>> c
16
```

Numerical expressions

```
>>> a+b*c        ← 1
(12.424777800000001+9.86960406437476j)
>>> a**2+b**2
18.869604064374762
>>> a**2-b**2+c  ← 2
(2.1303959356252395+3.1415926j)
```

1. Python's numerical expressions are pretty standard
2. Different (but compatible) data types are automatically converted whenever possible

Conditional expressions

```
>>> a = 7
>>> b = 9
>>> a if a>b else b          ← 1
9
>>> c = a+(a if a>b else b)  ← 2
>>> c
16
```

1. A conditional expression

Data types and expressions -2

Numerical expressions

```
>>> a+b*c          ← 1
(12.424777800000001+9.86960406437476j)
>>> a**2+b**2
18.869604064374762
>>> a**2-b**2+c    ← 2
(2.130395935625239 5+3.1415926j)
```

1. Python's numerical expressions are pretty standard
2. Different (but compatible) data types are automatically converted whenever possible

Conditional expressions

```
>>> a = 7
>>> b = 9
>>> a if a>b else b        ← 1
9
>>> c = a+(a if a>b else b)  ← 2
>>> c
16
```

1. A conditional expression
2. Conditional expressions can be used as any other Python expression

Numerical expressions

```
>>> a+b*c          ← 1
(12.424777800000001+9.86960406437476j)
>>> a**2+b**2
18.869604064374762
>>> a**2-b**2+c    ← 2
(2.1303959356252395+3.1415926j)
```

1. Python's numerical expressions are pretty standard
2. Different (but compatible) data types are automatically converted whenever possible

Conditional expressions

```
>>> a = 7
>>> b = 9
>>> a if a>b else b          ← 1
9
>>> c = a+(a if a>b else b)  ← 2
>>> c
16
```

1. A conditional expression
2. Conditional expressions can be used as any other Python expression

→

Boolean type and logical expressions

```
>>> a = 3
>>> b = 9
>>> a == b
False
>>> a > b
False
>>> a < b
True
>>> c = 7
>>> c < b and c > a
True
```

Boolean type and logical expressions

```
>>> a = 3
>>> b = 9
>>> a == b
False   ← 1
>>> a > b
False
>>> a < b
True   ← 1
>>> c = 7
>>> c < b and c > a
True
```

Data types and expressions -3

Boolean type and logical expressions

```
>>> a = 3
>>> b = 9
>>> a == b      ← 2,3
False     ← 1
>>> a > b       ← 2
False
>>> a < b       ← 2
True      ← 1
>>> c = 7
>>> c < b and c > a     ← 2
True
```

Boolean type and logical expressions

```
>>> a = 3
>>> b = 9
>>> a == b    ← 2,3
False    ← 1
>>> a > b    ← 2
False
>>> a < b    ← 2
True    ← 1
>>> c = 7
>>> c < b and c > a    ← 2
True
```

1. Python provides a boolean type with values: `True` and `False`

Boolean type and logical expressions

```
>>> a = 3
>>> b = 9
>>> a == b      ⟵ 2,3
False   ⟵ 1
>>> a > b       ⟵ 2
False
>>> a < b       ⟵ 2
True    ⟵ 1
>>> c = 7
>>> c < b and c > a     ⟵ 2
True
```

1. Python provides a boolean type with values: `True` and `False`
2. Python's logical expressions are pretty standard

Data types and expressions -3

Boolean type and logical expressions

```
>>> a = 3
>>> b = 9
>>> a == b   ← 2,3
False        ← 1
>>> a > b    ← 2
False
>>> a < b    ← 2
True         ← 1
>>> c = 7
>>> c < b and c > a   ← 2
True
```

1. Python provides a boolean type with values: `True` and `False`
2. Python's logical expressions are pretty standard
3. Note the difference between **=** (assignment symbol) and **==** (logical equality operator)

Data types and expressions -3

Boolean type and logical expressions

```
>>> a = 3
>>> b = 9
>>> a == b        ← 2,3
False         ← 1
>>> a > b         ← 2
False
>>> a < b         ← 2
True          ← 1
>>> c = 7
>>> c < b and c > a       ← 2
True
```

1. Python provides a boolean type with values: `True` and `False`
2. Python's logical expressions are pretty standard
3. Note the difference between **=** (assignment symbol) and **==** (logical equality operator)

→

Data types and expressions -4

Other Python types may be interpreted as True/False boolean values in logical expressions.

Note: the `bool()` function evaluates the boolean value of its argument...

Other Python types may be interpreted as True/False boolean values in logical expressions.

Note: the `bool()` function evaluates the boolean value of its argument...

```
>>> bool(0)
False
>>> bool(127)
True
>>> bool(0.0)
False
>>> bool(0.0000001)
True
>>> bool(complex(0,0))
False
>>> bool(complex(0,0.01))
True
>>> bool("")
False
>>> bool("abc")
True
>>> bool([])
False
>>> bool([1,2,3])
True
```

Data types and expressions -4

Other Python types may be interpreted as True/False boolean values in logical expressions.

Note: the `bool()` function evaluates the boolean value of its argument...

```
>>> bool(0)                  ←  1
False
>>> bool(127)                ←  1
True
>>> bool(0.0)                ←  1
False
>>> bool(0.0000001)          ←  1
True
>>> bool(complex(0,0))       ←  1
False
>>> bool(complex(0,0.01))    ←  1
True
>>> bool("")
False
>>> bool("abc")
True
>>> bool([])
False
>>> bool([1,2,3])
True
```

Data types and expressions -4

Other Python types may be interpreted as True/False boolean values in logical expressions.

Note: the `bool()` function evaluates the boolean value of its argument...

```
>>> bool(0)          ← 1
False
>>> bool(127)        ← 1
True
>>> bool(0.0)        ← 1
False
>>> bool(0.0000001)  ← 1
True
>>> bool(complex(0,0))  ← 1
False
>>> bool(complex(0,0.01))  ← 1
True
>>> bool("")         ← 2
False
>>> bool("abc")      ← 2
True
>>> bool([])         ← 2
False
>>> bool([1,2,3])    ← 2
True
```

Other Python types may be interpreted as True/False boolean values in logical expressions.

Note: the `bool()` function evaluates the boolean value of its argument...

```
>>> bool(0)             ← 1
False
>>> bool(127)           ← 1
True
>>> bool(0.0)           ← 1
False
>>> bool(0.0000001)     ← 1
True
>>> bool(complex(0,0))  ← 1
False
>>> bool(complex(0,0.01))  ← 1
True
>>> bool("")            ← 2
False
>>> bool("abc")         ← 2
True
>>> bool([])            ← 2
False
>>> bool([1,2,3])       ← 2
True
```

1. A numerical value is `False` if zero, `True` otherwise

Other Python types may be interpreted as True/False boolean values in logical expressions.

Note: the `bool()` function evaluates the boolean value of its argument...

```
>>> bool(0)            ← 1
False
>>> bool(127)          ← 1
True
>>> bool(0.0)          ← 1
False
>>> bool(0.0000001)    ← 1
True
>>> bool(complex(0,0))    ← 1
False
>>> bool(complex(0,0.01)) ← 1
True
>>> bool("")           ← 2
False
>>> bool("abc")        ← 2
True
>>> bool([])           ← 2
False
>>> bool([1,2,3])      ← 2
True
```

1. A numerical value is `False` if zero, `True` otherwise
2. A string (or any other kind of “collection”) evaluates to `False` if empty, to `True` if not empty

Other Python types may be interpreted as True/False boolean values in logical expressions.

Note: the `bool()` function evaluates the boolean value of its argument...

```
>>> bool(0)                  ← 1
False
>>> bool(127)                ← 1
True
>>> bool(0.0)                ← 1
False
>>> bool(0.0000001)          ← 1
True
>>> bool(complex(0,0))       ← 1
False
>>> bool(complex(0,0.01))    ← 1
True
>>> bool("")                 ← 2
False
>>> bool("abc")              ← 2
True
>>> bool([])                 ← 2
False
>>> bool([1,2,3])            ← 2
True
```

1. A numerical value is `False` if zero, `True` otherwise
2. A string (or any other kind of "collection") evaluates to `False` if empty, to `True` if not empty

→

String type and expressions

```
>>> a = "The quick brown fox"
>>> b = "jumps over the lazy dog"
>>> a+" "+b
'The quick brown fox jumps over the lazy dog'
>>> a > b
False
>>> a[2], b[7]
('e', 'v')
>>> a.upper()
'THE QUICK BROWN FOX'
>>> b.find("the")
11
>>> a.endswith("fox")
True
```

String type and expressions

```
>>> a = "The quick brown fox"
>>> b = "jumps over the lazy dog"
>>> a+" "+b            ← 1
'The quick brown fox jumps over the lazy dog'
>>> a > b              ← 2
False
>>> a[2], b[7]         ← 3
('e', 'v')
>>> a.upper()
'THE QUICK BROWN FOX'
>>> b.find("the")      ← 4
11
>>> a.endswith("fox")
True
```

String type and expressions

```
>>> a = "The quick brown fox"
>>> b = "jumps over the lazy dog"
>>> a+" "+b                          ← 1
'The quick brown fox jumps over the lazy dog'
>>> a > b                            ← 2
False
>>> a[2], b[7]                       ← 3
('e', 'v')
>>> a.upper()                   |
'THE QUICK BROWN FOX'           |
>>> b.find("the")               | ← 4
11                              |
>>> a.endswith("fox")           |
True                            |
```

1. In string expressions the symbol **+** indicates concatenation

String type and expressions

```
>>> a = "The quick brown fox"
>>> b = "jumps over the lazy dog"
>>> a+" "+b              ← 1
'The quick brown fox jumps over the lazy dog'
>>> a > b               ← 2
False
>>> a[2], b[7]            ← 3
('e', 'v')
>>> a.upper()           |
'THE QUICK BROWN FOX'   |
>>> b.find("the")       | ← 4
11                      |
>>> a.endswith("fox")   |
True                    |
```

1. In string expressions the symbol **+** indicates concatenation
2. The symbols $<$, $>$ evaluate the lexical order

Data types and expressions -5

String type and expressions

```
>>> a = "The quick brown fox"
>>> b = "jumps over the lazy dog"
>>> a+" "+b            ← 1
'The quick brown fox jumps over the lazy dog'
>>> a > b              ← 2
False
>>> a[2], b[7]         ← 3
('e', 'v')
>>> a.upper()
'THE QUICK BROWN FOX'
>>> b.find("the")      ← 4
11
>>> a.endswith("fox")
True
```

1. In string expressions the symbol **+** indicates concatenation
2. The symbols <, > evaluate the lexical order
3. Strings are also “collections of characters”

Data types and expressions -5

String type and expressions

```
>>> a = "The quick brown fox"
>>> b = "jumps over the lazy dog"
>>> a+" "+b            ← 1
'The quick brown fox jumps over the lazy dog'
>>> a > b              ← 2
False
>>> a[2], b[7]         ← 3
('e', 'v')
>>> a.upper()
'THE QUICK BROWN FOX'
>>> b.find("the")      ← 4
11
>>> a.endswith("fox")
True
```

1. In string expressions the symbol **+** indicates concatenation
2. The symbols <, > evaluate the lexical order
3. Strings are also “collections of characters”
4. Various `methods` operate on strings

Data types and expressions -5

String type and expressions

```
>>> a = "The quick brown fox"
>>> b = "jumps over the lazy dog"
>>> a+" "+b          ← 1
'The quick brown fox jumps over the lazy dog'
>>> a > b            ← 2
False
>>> a[2], b[7]       ← 3
('e', 'v')
>>> a.upper()
'THE QUICK BROWN FOX'
>>> b.find("the")    ← 4
11
>>> a.endswith("fox")
True
```

1. In string expressions the symbol **+** indicates concatenation
2. The symbols <, > evaluate the lexical order
3. Strings are also “collections of characters”
4. Various `methods` operate on strings

> In Python strings are represented with **unicode** characters, which allows the support for most languages in the world.

String type and expressions

```
>>> a = "The quick brown fox"
>>> b = "jumps over the lazy dog"
>>> a+" "+b          <- 1
'The quick brown fox jumps over the lazy dog'
>>> a > b            <- 2
False
>>> a[2], b[7]       <- 3
('e', 'v')
>>> a.upper()
'THE QUICK BROWN FOX'
>>> b.find("the")    <- 4
11
>>> a.endswith("fox")
True
```

1. In string expressions the symbol **+** indicates concatenation
2. The symbols $<$, $>$ evaluate the lexical order
3. Strings are also "collections of characters"
4. Various `methods` operate on strings

> In Python strings are represented with **unicode** characters, which allows the support for most languages in the world.

→

String formatting

String interpolation with % operator

Format specifications (%d, %s, ...) in the format string are substituted by corresponding values, properly converted

String formatting

String interpolation with % operator

Format specifications (%d, %s, ...) in the format string are substituted by corresponding values, properly converted

```
>>> a = 3.3
>>> b = 2
>>> c = a+b
>>> d = "%d %s %d is %d" % (a,"plus",b,c)
>>> d
'3 plus 2 is 5'
```

String formatting

String interpolation with % operator

Format specifications (%d, %s, ...) in the format string are substituted by corresponding values, properly converted

```
>>> a = 3.3
>>> b = 2
>>> c = a+b
>>> d = "%d %s %d is %d" % (a,"plus",b,c)
>>> d
'3 plus 2 is 5'
```

String formatting

String interpolation with % operator

Format specifications (%d, %s, ...) in the format string are substituted by corresponding values, properly converted

```
>>> a = 3.3
>>> b = 2
>>> c = a+b
>>> d = "%d %s %d is %d" % (a,"plus",b,c)
>>> d
'3 plus 2 is 5'
```

String interpolation with % operator

Format specifications (%d, %s, ...) in the format string are substituted by corresponding values, properly converted

```
>>> a = 3.3
>>> b = 2
>>> c = a+b
>>> d = "%d %s %d is %d" % (a,"plus",b,c)
>>> d
'3 plus 2 is 5'
```

String interpolation with % operator

Format specifications (%d, %s, ...) in the format string are substituted by corresponding values, properly converted

```
>>> a = 3.3
>>> b = 2
>>> c = a+b
>>> d = "%d %s %d is %d" % (a,"plus",b,c)
>>> d
'3 plus 2 is 5'
```

String formatting

String interpolation with % operator

Format specifications (%d, %s, ...) in the format string are substituted by corresponding values, properly converted

```
>>> a = 3.3
>>> b = 2
>>> c = a+b
>>> d = "%d %s %d is %d" % (a,"plus",b,c)
>>> d
'3 plus 2 is 5'
```

String interpolation with the `format()` method

```
>>> a = 3.3
>>> b = 2
>>> c = a+b
>>> "{0} {2} {1} is {result}".format(a, b, "plus", result=c)
'3.3 plus 2 is 5.3'
```

String formatting

String interpolation with % operator

Format specifications (%d, %s, ...) in the format string are substituted by corresponding values, properly converted

```
>>> a = 3.3
>>> b = 2
>>> c = a+b
>>> d = "%d %s %d is %d" % (a,"plus",b,c)
>>> d
'3 plus 2 is 5'
```

String interpolation with the `format()` method

```
>>> a = 3.3
>>> b = 2
>>> c = a+b
>>> "{0} {2} {1} is {result}".format(a, b, "plus", result=c)
'3.3 plus 2 is 5.3'
```

String formatting

String interpolation with % operator

Format specifications (%d, %s, ...) in the format string are substituted by corresponding values, properly converted

```
>>> a = 3.3
>>> b = 2
>>> c = a+b
>>> d = "%d %s %d is %d" % (a,"plus",b,c)
>>> d
'3 plus 2 is 5'
```

String interpolation with the `format()` method

```
>>> a = 3.3
>>> b = 2
>>> c = a+b
>>> "{0} {2} {1} is {result}".format(a, b, "plus", result=c)
'3.3 plus 2 is 5.3'
```

String formatting

String interpolation with % operator

Format specifications (%d, %s, ...) in the format string are substituted by corresponding values, properly converted

```
>>> a = 3.3
>>> b = 2
>>> c = a+b
>>> d = "%d %s %d is %d" % (a,"plus",b,c)
>>> d
'3 plus 2 is 5'
```

String interpolation with the `format()` method

```
>>> a = 3.3
>>> b = 2
>>> c = a+b
>>> "{0} {2} {1} is {result}".format(a, b, "plus", result=c)
'3.3 plus 2 is 5.3'
```

String formatting

String interpolation with % operator

Format specifications (%d, %s, ...) in the format string are substituted by corresponding values, properly converted

```
>>> a = 3.3
>>> b = 2
>>> c = a+b
>>> d = "%d %s %d is %d" % (a,"plus",b,c)
>>> d
'3 plus 2 is 5'
```

String interpolation with the `format()` method

```
>>> a = 3.3
>>> b = 2
>>> c = a+b
>>> "{0} {2} {1} is {result}".format(a, b, "plus", result=c)
'3.3 plus 2 is 5.3'
```

String formatting

String interpolation with % operator

Format specifications (%d, %s, ...) in the format string are substituted by corresponding values, properly converted

```
>>> a = 3.3
>>> b = 2
>>> c = a+b
>>> d = "%d %s %d is %d" % (a,"plus",b,c)
>>> d
'3 plus 2 is 5'
```

String interpolation with the `format()` method

```
>>> a = 3.3
>>> b = 2
>>> c = a+b
>>> "{0} {2} {1} is {result}".format(a, b, "plus", result=c)
'3.3 plus 2 is 5.3'
```

String interpolation with **f** strings

```
>>> a = 3.3
>>> b = 2
>>> c = a+b
>>> s = "plus"
>>> f"{a} {s} {b} is {c}"
'3.3 plus 2 is 5.3'
```

String formatting

String interpolation with % operator

Format specifications (%d, %s, ...) in the format string are substituted by corresponding values, properly converted

```
>>> a = 3.3
>>> b = 2
>>> c = a+b
>>> d = "%d %s %d is %d" % (a,"plus",b,c)
>>> d
'3 plus 2 is 5'
```

String interpolation with the `format()` method

```
>>> a = 3.3
>>> b = 2
>>> c = a+b
>>> "{0} {2} {1} is {result}".format(a, b, "plus", result=c)
'3.3 plus 2 is 5.3'
```

String interpolation with **f** strings

```
>>> a = 3.3
>>> b = 2
>>> c = a+b
>>> s = "plus"
>>> f"{a} {s} {b} is {c}"
'3.3 plus 2 is 5.3'
```

← 1

String formatting

String interpolation with % operator

Format specifications (%d, %s, ...) in the format string are substituted by corresponding values, properly converted

```
>>> a = 3.3
>>> b = 2
>>> c = a+b
>>> d = "%d %s %d is %d" % (a,"plus",b,c)
>>> d
'3 plus 2 is 5'
```

String interpolation with the `format()` method

```
>>> a = 3.3
>>> b = 2
>>> c = a+b
>>> "{0} {2} {1} is {result}".format(a, b, "plus", result=c)
'3.3 plus 2 is 5.3'
```

String interpolation with **f** strings

```
>>> a = 3.3
>>> b = 2
>>> c = a+b
>>> s = "plus"
>>> f"{a} {s} {b} is {c}"   ← 1
'3.3 plus 2 is 5.3'
```

1. Here we use directly variable names

String interpolation with % operator

Format specifications (%d, %s, ...) in the format string are substituted by corresponding values, properly converted

```
>>> a = 3.3
>>> b = 2
>>> c = a+b
>>> d = "%d %s %d is %d" % (a,"plus",b,c)
>>> d
'3 plus 2 is 5'
```

String interpolation with the `format()` method

```
>>> a = 3.3
>>> b = 2
>>> c = a+b
>>> "{0} {2} {1} is {result}".format(a, b, "plus", result=c)
'3.3 plus 2 is 5.3'
```

String interpolation with **f** strings

```
>>> a = 3.3
>>> b = 2
>>> c = a+b
>>> s = "plus"
>>> f"{a} {s} {b} is {c}"     ← 1
'3.3 plus 2 is 5.3'
```

1. Here we use directly variable names

→

The *None* type

Python provides a **None** type to express ”non existent” or ”undefined” values

The *None* type

Python provides a **None** type to express ”non existent” or ”undefined” values

```
>>> a = None
>>> type(a)
<class 'NoneType'>

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

>>> "abc"+a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not NoneType

>>> bool(a)
False
```

The *None* type

Python provides a **None** type to express ”non existent” or ”undefined” values

```
>>> a = None     ← 1
>>> type(a)
<class 'NoneType'>

>>> a+1     ← 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'

>>> "abc"+a     ← 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not NoneType

>>> bool(a)     ← 4
False
```

Data types and expressions -6

The *None* type

Python provides a **None** type to express ”non existent” or ”undefined” values

```
>>> a = None        ← 1
>>> type(a)
<class 'NoneType'>

>>> a+1      ← 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'

>>> "abc"+a      ← 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not NoneType

>>> bool(a)      ← 4
False
```

1. None is a constant of type `None`

The *None* type

Python provides a **None** type to express ”non existent” or ”undefined” values

```
>>> a = None        ← 1
>>> type(a)
<class 'NoneType'>

>>> a+1     ← 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'

>>> "abc"+a     ← 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not NoneType

>>> bool(a)     ← 4
False
```

1. None is a constant of type `None`
2. None is not an integer ...

The *None* type

Python provides a **None** type to express ”non existent” or ”undefined” values

```
>>> a = None        ← 1
>>> type(a)
<class 'NoneType'>

>>> a+1        ← 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'

>>> "abc"+a        ← 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not NoneType

>>> bool(a)        ← 4
False
```

1. None is a constant of type `None`
2. None is not an integer ...
3. None is not a string ...

The *None* type

Python provides a **None** type to express ”non existent” or ”undefined” values

```
>>> a = None    ← 1
>>> type(a)
<class 'NoneType'>

>>> a+1    ← 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'

>>> "abc"+a    ← 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not NoneType

>>> bool(a)    ← 4
False
```

1. None is a constant of type `None`
2. None is not an integer ...
3. None is not a string ...
4. None is logically *False* ...

The *None* type

Python provides a **None** type to express ”non existent” or ”undefined” values

```
>>> a = None        ← 1
>>> type(a)
<class 'NoneType'>

>>> a+1        ← 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'

>>> "abc"+a        ← 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not NoneType

>>> bool(a)        ← 4
False
```

1. None is a constant of type `None`
2. None is not an integer ...
3. None is not a string ...
4. None is logically *False* ...

→

Conditional statement

Conditional statement

```
>>> a = 7
>>> b = 9
>>> if a>b:
...     print(a,">",b)
...     c = a+a
... elif a<b:
...     print(b,">",a)
...     c = a+b
... else:
...     print(a,"=",b)
...     c = a*2
...
9 > 7
>>> c
16
```

Conditional statement

```
>>> a = 7
>>> b = 9
>>> if a>b:          ← 1, 2
...     print(a,">",b)
...     c = a+a       ← 3
... elif a<b:
...     print(b,">",a)
...     c = a+b       ← 3
... else:
...     print(a,"=",b)
...     c = a*2       ← 3
...
9 > 7
>>> c
16
```

Conditionals and loops -1

Conditional statement

```
>>> a = 7
>>> b = 9
>>> if a>b:          ← 1, 2
...     print(a,">",b)
...     c = a+a          ← 3
... elif a<b:
...     print(b,">",a)
...     c = a+b          ← 3
... else:
...     print(a,"=",b)
...     c = a*2          ← 3
...
9 > 7
>>> c
16
```

1. Conditional statements use keywords: **if**, **elif**, **else**

Conditional statement

```
>>> a = 7
>>> b = 9
>>> if a>b:          ← 1, 2
...     print(a,">",b)
...     c = a+a      ← 3
... elif a<b:
...     print(b,">",a)
...     c = a+b      ← 3
... else:
...     print(a,"=",b)
...     c = a*2      ← 3
...
9 > 7
>>> c
16
```

1. Conditional statements use keywords: **if**, **elif**, **else**
2. Note the colon (**:**) at the end of the conditional clause

Conditionals and loops -1

Conditional statement

```
>>> a = 7
>>> b = 9
>>> if a>b:              ← 1, 2
...     print(a,">",b)
...     c = a+a          ← 3
... elif a<b:
...     print(b,">",a)
...     c = a+b          ← 3
... else:
...     print(a,"=",b)
...     c = a*2          ← 3
...
9 > 7
>>> c
16
```

1. Conditional statements use keywords: **if**, **elif**, **else**
2. Note the colon (**:**) at the end of the conditional clause
3. Conditional blocks are <u>indented</u>

Conditionals and loops -1

Conditional statement

```
>>> a = 7
>>> b = 9
>>> if a>b:          ← 1, 2
...     print(a,">",b)
...     c = a+a       ← 3
... elif a<b:
...     print(b,">",a)
...     c = a+b       ← 3
... else:
...     print(a,"=",b)
...     c = a*2       ← 3
...
9 > 7
>>> c
16
```

1. Conditional statements use keywords: **if**, **elif**, **else**
2. Note the colon (**:**) at the end of the conditional clause
3. Conditional blocks are <u>indented</u>

In Python text alignment is meaningful!

The amount of space before statements is arbitrary, provided it is the same for all the statements of the block.

Conditionals and loops -1

Conditional statement

```
>>> a = 7
>>> b = 9
>>> if a>b:     ←  1, 2
...     print(a,">",b)
...     c = a+a      ←  3
... elif a<b:
...     print(b,">",a)
...     c = a+b      ←  3
... else:
...     print(a,"=",b)
...     c = a*2      ←  3
...
9 > 7
>>> c
16
```

1. Conditional statements use keywords: **if**, **elif**, **else**
2. Note the colon (**:**) at the end of the conditional clause
3. Conditional blocks are indented

> In Python text alignment is meaningful!
>
> The amount of space before statements is arbitrary, provided it is the same for all the statements of the block.

→

The while loop

The loop block is repeated until the condition becomes `False`

```
>>> a = 5
>>> while a>1:
...     print("a =",a)
...     a -= 1
...
a = 5
a = 4
a = 3
a = 2
>>>
```

Conditionals and loops -2

The `while` loop

The loop block is repeated until the condition becomes `False`

```
>>> a = 5
>>> while a>1:
...     print("a =",a)    ← 1
...     a -= 1
...
a = 5
a = 4
a = 3
a = 2
>>>
```

The while loop

The loop block is repeated until the condition becomes `False`

```
>>> a = 5
>>> while a>1:
...     print("a =",a)      ← 1
...     a -= 1
...
a = 5
a = 4
a = 3
a = 2
>>>
```

1. Again, the loop block is indented

The `while` loop

The loop block is repeated until the condition becomes `False`

```
>>> a = 5
>>> while a>1:
...    print("a =",a)    ← 1
...    a -= 1
...
a = 5
a = 4
a = 3
a = 2
>>>
```

1. Again, the loop block is indented

The `for` loop

Python's `for` loop iterates on the elements of a collection

```
>>> for c in "Hi there!":
...   print(c)
...
H
i

t
h
e
r
e
!
>>>
```

The `while` loop

> The loop block is repeated until the condition becomes `False`

```
>>> a = 5
>>> while a>1:
...     print("a =",a)     ← 1
...     a -= 1
...
a = 5
a = 4
a = 3
a = 2
>>>
```

1. Again, the loop block is indented

The `for` loop

> Python's `for` loop iterates on the elements of a collection

```
>>> for c in "Hi there!":
...   print(c)
...
H
i

t
h
e
r
e
!
>>>
```

The break statement

```
>>> for c in "Hi there!":
...   if c==" ":
...      break
...   print(c)
...
H
i
>>>
```

The break statement

```
>>> for c in "Hi there!":
...   if c==" ":
...     break ⟵
...   print(c)
...
H
i
>>>
```

The break statement

```
>>> for c in "Hi there!":
...   if c==" ":
...      break ←
...   print(c)
...
H
i
>>>
```

- Loops can be interrupted before their “natural” end with the `break` statement

The `break` statement

```
>>> for c in "Hi there!":
...   if c==" ":
...      break
...   print(c)
...
H
i
>>>
```

- Loops can be interrupted before their “natural” end with the `break` statement

The `continue` statement

```
>>> for c in "Hi there!":
...  if c in " !":
...    continue
...  print(c)
...
H
i
t
h
e
r
e
>>>
```

The break statement

```
>>> for c in "Hi there!":
...   if c==" ":
...      break ←
...   print(c)
...
H
i
>>>
```

- Loops can be interrupted before their “natural” end with the `break` statement

The continue statement

```
>>> for c in "Hi there!":
...  if c in " !": ← 2
...    continue  ← 1
...  print(c)
...
H
i
t
h
e
r
e
>>>
```

The break statement

```
>>> for c in "Hi there!":
...    if c==" ":
...       break ←
...    print(c)
...
H
i
>>>
```

- Loops can be interrupted before their “natural” end with the `break` statement

The continue statement

```
>>> for c in "Hi there!":
...   if c in " !":  ← 2
...     continue   ← 1
...   print(c)
...
H
i
t
h
e
r
e
>>>
```

1. The normal flux of a loop can be modified by the `continue` statement, which jumps to the next iteration

The `break` statement

```
>>> for c in "Hi there!":
...   if c==" ":
...      break  ⟵
...   print(c)
...
H
i
>>>
```

- Loops can be interrupted before their “natural” end with the `break` statement

The `continue` statement

```
>>> for c in "Hi there!":
...  if c in " !":  ⟵ 2
...    continue  ⟵ 1
...  print(c)
...
H
i
t
h
e
r
e
>>>
```

1. The normal flux of a loop can be modified by the `continue` statement, which jumps to the next iteration
2. Note another use of the keyword **in**

The break statement

```
>>> for c in "Hi there!":
...    if c==" ":
...       break  ⟵
...    print(c)
...
H
i
>>>
```

- Loops can be interrupted before their “natural” end with the `break` statement

The continue statement

```
>>> for c in "Hi there!":
...   if c in " !":  ⟵ 2
...     continue   ⟵ 1
...   print(c)
...
H
i
t
h
e
r
e
>>>
```

1. The normal flux of a loop can be modified by the `continue` statement, which jumps to the next iteration
2. Note another use of the keyword **in**

⟶

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)
>>> text = "Hi there!"
```

Collections

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

Collections

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

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

Collections

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

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

Collections

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

1. *tuple*: non mutable collection of objects, possibly different in type. Referenced by **index**.
2. *list*: mutable collection of objects, possibly different in type. Referenced by **index**
3. *dictionary*: mutable collection of objects, possibly different in type. Referenced by **key**

Collections

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

1. *tuple*: non mutable collection of objects, possibly different in type. Referenced by **index**.
2. *list*: mutable collection of objects, possibly different in type. Referenced by **index**
3. *dictionary*: mutable collection of objects, possibly different in type. Referenced by **key**
4. *set*: mutable collection of unique objects, (*frozenset*: non mutable). Supports usual operations on sets

Collections

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

1. *tuple*: non mutable collection of objects, possibly different in type. Referenced by **index**.
2. *list*: mutable collection of objects, possibly different in type. Referenced by **index**
3. *dictionary*: mutable collection of objects, possibly different in type. Referenced by **key**
4. *set*: mutable collection of unique objects, (*frozenset*: non mutable). Supports usual operations on sets
5. *string*: non mutable collection of characters

Collections

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

1. *tuple*: non mutable collection of objects, possibly different in type. Referenced by **index**.
2. *list*: mutable collection of objects, possibly different in type. Referenced by **index**
3. *dictionary*: mutable collection of objects, possibly different in type. Referenced by **key**
4. *set*: mutable collection of unique objects, (*frozenset*: non mutable). Supports usual operations on sets
5. *string*: non mutable collection of characters

→

Collections -2

From the Manual:

Operation	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**

Collections -2

From the Manual:

Operation	Result	
`x in s`	*True* if any element of **s** is equal to **x**, else *False*	-1
`x not in s`	*False* if any element of **s** is equal to **x**, else *True*	
`s+t`	concatenation of **s** and **t**	-2
`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**	

From the Manual:

Operation	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**

-1

-2

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

From the Manual:

Operation	Result	
`x in s`	*True* if any element of **s** is equal to **x**, else *False*	-1
`x not in s`	*False* if any element of **s** is equal to **x**, else *True*	
`s+t`	concatenation of **s** and **t**	-2
`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**	

1. What's the meaning if **s** is a dictionary?
2. We have seen already the **+** sign with the meaning of concatenation

Collections -2

From the Manual:

Operation	Result	
`x in s`	*True* if any element of **s** is equal to **x**, else *False*	-1
`x not in s`	*False* if any element of **s** is equal to **x**, else *True*	
`s+t`	concatenation of **s** and **t**	-2
`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**	

1. What's the meaning if **s** is a dictionary?
2. We have seen already the **+** sign with the meaning of concatenation

→

Comprehension

```
>>> atuple
(1, 2, 4, 'five', 6.0, -7, 'VIII')
>>> newlist = [x*x for x in atuple if type(x) in (int, float)]
>>> newlist
[1, 4, 16, 36.0, 49]
```

Comprehension

```
>>> atuple
(1, 2, 4, 'five', 6.0, -7, 'VIII')
>>> newlist = [x*x for x in atuple if type(x) in (int, float)]   ← 1
>>> newlist
[1, 4, 16, 36.0, 49]   ← 2
```

Collections -3

Comprehension

```
>>> atuple
(1, 2, 4, 'five', 6.0, -7, 'VIII')
>>> newlist = [x*x for x in atuple if type(x) in (int, float)] ⟵ 1
>>> newlist
[1, 4, 16, 36.0, 49] ⟵ 2
```

1. Powerful syntax to create collections from other collections

Comprehension

```
>>> atuple
(1, 2, 4, 'five', 6.0, -7, 'VIII')
>>> newlist = [x*x for x in atuple if type(x) in (int, float)] ← 1
>>> newlist
[1, 4, 16, 36.0, 49] ← 2
```

1. Powerful syntax to create collections from other collections
2. Here we created a list from a tuple

Comprehension

```
>>> atuple
(1, 2, 4, 'five', 6.0, -7, 'VIII')
>>> newlist = [x*x for x in atuple if type(x) in (int, float)]   ⟵ 1
>>> newlist
[1, 4, 16, 36.0, 49]   ⟵ 2
```

1. Powerful syntax to create collections from other collections
2. Here we created a list from a tuple

```
>>> kk = ("one", "two", "three")
>>> vv = (1, 2, 3)
>>> newdict = {k: v for k, v in zip(kk, vv)}
>>> newdict
{'one': 1, 'two': 2, 'three': 3}
```

Comprehension

```
>>> atuple
(1, 2, 4, 'five', 6.0, -7, 'VIII')
>>> newlist = [x*x for x in atuple if type(x) in (int, float)]   ← 1
>>> newlist
[1, 4, 16, 36.0, 49]   ← 2
```

1. Powerful syntax to create collections from other collections
2. Here we created a list from a tuple

```
>>> kk = ("one", "two", "three")
>>> vv = (1, 2, 3)
>>> newdict = {k: v for k, v in zip(kk, vv)}   ← 2
>>> newdict
{'one': 1, 'two': 2, 'three': 3}   ← 1
```

Comprehension

```
>>> atuple
(1, 2, 4, 'five', 6.0, -7, 'VIII')
>>> newlist = [x*x for x in atuple if type(x) in (int, float)]  ← 1
>>> newlist
[1, 4, 16, 36.0, 49]  ← 2
```

1. Powerful syntax to create collections from other collections
2. Here we created a list from a tuple

```
>>> kk = ("one", "two", "three")
>>> vv = (1, 2, 3)
>>> newdict = {k: v for k, v in zip(kk, vv)}  ← 2
>>> newdict
{'one': 1, 'two': 2, 'three': 3}  ← 1
```

1. Here we created a dictionary

Comprehension

```
>>> atuple
(1, 2, 4, 'five', 6.0, -7, 'VIII')
>>> newlist = [x*x for x in atuple if type(x) in (int, float)]  ← 1
>>> newlist
[1, 4, 16, 36.0, 49]  ← 2
```

1. Powerful syntax to create collections from other collections
2. Here we created a list from a tuple

```
>>> kk = ("one", "two", "three")
>>> vv = (1, 2, 3)
>>> newdict = {k: v for k, v in zip(kk, vv)}  ← 2
>>> newdict
{'one': 1, 'two': 2, 'three': 3}  ← 1
```

1. Here we created a dictionary
2. The `zip()` function:

```
>>> list(zip(kk, vv))
[('one', 1), ('two', 2), ('three', 3)]
```

Comprehension

```
>>> atuple
(1, 2, 4, 'five', 6.0, -7, 'VIII')
>>> newlist = [x*x for x in atuple if type(x) in (int, float)]  ⟵ 1
>>> newlist
[1, 4, 16, 36.0, 49]  ⟵ 2
```

1. Powerful syntax to create collections from other collections
2. Here we created a list from a tuple

```
>>> kk = ("one", "two", "three")
>>> vv = (1, 2, 3)
>>> newdict = {k: v for k, v in zip(kk, vv)}  ⟵ 2
>>> newdict
{'one': 1, 'two': 2, 'three': 3}  ⟵ 1
```

1. Here we created a dictionary
2. The `zip()` function:

   ```
   >>> list(zip(kk, vv))
   [('one', 1), ('two', 2), ('three', 3)]
   ```

→

Python programs

Up to now we've used Python's interactive mode to test our code

Python programs

Up to now we've used Python's interactive mode to test our code

Usually instead Python code is written into files to be used many times or to be shared with others

Python programs

Up to now we've used Python's interactive mode to test our code

Usually instead Python code is written into files to be used many times or to be shared with others

So, let's suppose we have a file `loop.py` containing the code shown below

file: `loop.py`

```
# Example code

for c in "Hi there!":
    print(c, end=" ")
print()
```

Python programs

Up to now we've used Python's interactive mode to test our code

Usually instead Python code is written into files to be used many times or to be shared with others

So, let's suppose we have a file `loop.py` containing the code shown below

file: `loop.py`

```
# Example code  ⟵  1

for c in "Hi there!":       2
    print(c, end=" ")
print()
```

Python programs

Up to now we've used Python's interactive mode to test our code

Usually instead Python code is written into files to be used many times or to be shared with others

So, let's suppose we have a file `loop.py` containing the code shown below

file: `loop.py`

```
# Example code ⟵  1

for c in "Hi there!":        2
    print(c, end=" ")
print()
```

1. This is a comment line

Python programs

Up to now we've used Python's interactive mode to test our code

Usually instead Python code is written into files to be used many times or to be shared with others

So, let's suppose we have a file `loop.py` containing the code shown below

file: `loop.py`

```
# Example code        ← 1

for c in "Hi there!":        2
    print(c, end=" ")
print()
```

1. This is a comment line
2. **Note:** in the slide different elements of the language are shown in different colors. The same happens when using "language aware" editors to write programs

Python programs

Up to now we've used Python's interactive mode to test our code

Usually instead Python code is written into files to be used many times or to be shared with others

So, let's suppose we have a file `loop.py` containing the code shown below

file: `loop.py`

```
# Example code  ⟵  1

for c in "Hi there!":        2
    print(c, end=" ")
print()
```

1. This is a comment line
2. **Note:** in the slide different elements of the language are shown in different colors. The same happens when using "language aware" editors to write programs

You execute your program as follows:

```
$ python loop.py
H i   t h e r e !
```

Python programs

Up to now we've used Python's interactive mode to test our code

Usually instead Python code is written into files to be used many times or to be shared with others

So, let's suppose we have a file `loop.py` containing the code shown below

file: `loop.py`

```
# Example code   <-- 1

for c in "Hi there!":       2
    print(c, end=" ")
print()
```

1. This is a comment line
2. **Note:** in the slide different elements of the language are shown in different colors. The same happens when using "language aware" editors to write programs

You execute your program as follows:

```
$ python loop.py
H i   t h e r e !
```

Functions and arguments -1

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)
```

Functions and arguments -1

file: `funny.py` (function **definition**)

```
def funny(a, b, default=3.1415926, toint=False):  ⟵ 1, 2
    if toint:
        cvt = lambda x: int(x)  ⟵ 3
    else:
        cvt = lambda x: x  ⟵ 3
    if a > b:
        return cvt(a)  ⟵ 4
    elif a < b:
        return cvt(b)  ⟵ 4
    return cvt(default)  ⟵ 4
```

Functions and arguments -1

file: `funny.py` (function **definition**)

```
def funny(a, b, default=3.1415926, toint=False):  ⟵ 1, 2
    if toint:
        cvt = lambda x: int(x)  ⟵ 3
    else:
        cvt = lambda x: x  ⟵ 3
    if a > b:
        return cvt(a)  ⟵ 4
    elif a < b:
        return cvt(b)  ⟵ 4
    return cvt(default)  ⟵ 4
```

1. a, b: **positional** arguments

Functions and arguments -1

file: funny.py (function **definition**)

```
def funny(a, b, default=3.1415926, toint=False):  ⟵ 1, 2
    if toint:
        cvt = lambda x: int(x)  ⟵ 3
    else:
        cvt = lambda x: x  ⟵ 3
    if a > b:
        return cvt(a)  ⟵ 4
    elif a < b:
        return cvt(b)  ⟵ 4
    return cvt(default)  ⟵ 4
```

1. a, b: **positional** arguments
2. `default`, `toint`: **named** (optional) arguments

file: `funny.py` (function **definition**)

```
def funny(a, b, default=3.1415926, toint=False):  ⟵ 1, 2
    if toint:
        cvt = lambda x: int(x)  ⟵ 3
    else:
        cvt = lambda x: x  ⟵ 3
    if a > b:
        return cvt(a)  ⟵ 4
    elif a < b:
        return cvt(b)  ⟵ 4
    return cvt(default)  ⟵ 4
```

1. `a, b`: **positional** arguments
2. `default, toint`: **named** (optional) arguments
3. `lambda`: unnamed function

Functions and arguments -1

file: funny.py (function **definition**)

```
def funny(a, b, default=3.1415926, toint=False): ⟵ 1, 2
    if toint:
        cvt = lambda x: int(x) ⟵ 3
    else:
        cvt = lambda x: x ⟵ 3
    if a > b:
        return cvt(a) ⟵ 4
    elif a < b:
        return cvt(b) ⟵ 4
    return cvt(default) ⟵ 4
```

1. `a, b`: **positional** arguments
2. `default`, `toint`: **named** (optional) arguments
3. `lambda`: unnamed function
4. The `return` statement

Functions and arguments -1

file: `funny.py` (function **definition**)

```
def funny(a, b, default=3.1415926, toint=False):   <- 1, 2
    if toint:
        cvt = lambda x: int(x)   <- 3
    else:
        cvt = lambda x: x   <- 3
    if a > b:
        return cvt(a)   <- 4
    elif a < b:
        return cvt(b)   <- 4
    return cvt(default)   <- 4
```

1. `a`, `b`: **positional** arguments
2. `default`, `toint`: **named** (optional) arguments
3. `lambda`: unnamed function
4. The `return` statement

→

Functions and arguments -2

How to use functions (function **call**):

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'
```

How to use functions (function **call**):

```
>>> from funny import funny     ← 1
>>> funny(1.0, 2.0)     ← 2
2.0
>>> funny(1.0, 1.0)     ← 2
3.1415926
>>> funny(1.0, 1.0, 6.2831852, toint=True)     ← 3
6
>>> funny(1.0, 1.0, toint=True, default=6.2831852)     ← 4
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'     ← 5
```

Functions and arguments -2

How to use functions (function **call**):

```
>>> from funny import funny                                    ← 1
>>> funny(1.0, 2.0)                                            ← 2
2.0
>>> funny(1.0, 1.0)                                            ← 2
3.1415926
>>> funny(1.0, 1.0, 6.2831852, toint=True)                     ← 3
6
>>> funny(1.0, 1.0, toint=True, default=6.2831852)             ← 4
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'  ← 5
```

1. To use a function defined in a file you must first **import** it.

How to use functions (function **call**):

```
>>> from funny import funny           ← 1
>>> funny(1.0, 2.0)          ← 2
2.0
>>> funny(1.0, 1.0)          ← 2
3.1415926
>>> funny(1.0, 1.0, 6.2831852, toint=True)          ← 3
6
>>> funny(1.0, 1.0, toint=True, default=6.2831852)          ← 4
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'          ← 5
```

1. To use a function defined in a file you must first **import** it.
2. Call with only positional arguments

Functions and arguments -2

How to use functions (function **call**):

```
>>> from funny import funny  ← 1
>>> funny(1.0, 2.0)  ← 2
2.0
>>> funny(1.0, 1.0)  ← 2
3.1415926
>>> funny(1.0, 1.0, 6.2831852, toint=True)  ← 3
6
>>> funny(1.0, 1.0, toint=True, default=6.2831852)  ← 4
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'  ← 5
```

1. To use a function defined in a file you must first **import** it.
2. Call with only positional arguments
3. Call with positional arguments, one optional argument (specified by position), one optional argument (specified by name)

How to use functions (function **call**):

```
>>> from funny import funny  ← 1
>>> funny(1.0, 2.0)  ← 2
2.0
>>> funny(1.0, 1.0)  ← 2
3.1415926
>>> funny(1.0, 1.0, 6.2831852, toint=True)  ← 3
6
>>> funny(1.0, 1.0, toint=True, default=6.2831852)  ← 4
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'  ← 5
```

1. To use a function defined in a file you must first **import** it.
2. Call with only positional arguments
3. Call with positional arguments, one optional argument (specified by position), one optional argument (specified by name)
4. Call with positional arguments, named arguments specified by name (order is irrelevant)

How to use functions (function **call**):

```
>>> from funny import funny          ⟵ 1
>>> funny(1.0, 2.0)          ⟵ 2
2.0
>>> funny(1.0, 1.0)          ⟵ 2
3.1415926
>>> funny(1.0, 1.0, 6.2831852, toint=True)          ⟵ 3
6
>>> funny(1.0, 1.0, toint=True, default=6.2831852)          ⟵ 4
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'          ⟵ 5
```

1. To use a function defined in a file you must first **import** it.
2. Call with only positional arguments
3. Call with positional arguments, one optional argument (specified by position), one optional argument (specified by name)
4. Call with positional arguments, named arguments specified by name (order is irrelevant)
5. Positional arguments are required

Functions and arguments -2

How to use functions (function **call**):

```
>>> from funny import funny  ← 1
>>> funny(1.0, 2.0)  ← 2
2.0
>>> funny(1.0, 1.0)  ← 2
3.1415926
>>> funny(1.0, 1.0, 6.2831852, toint=True)  ← 3
6
>>> funny(1.0, 1.0, toint=True, default=6.2831852)  ← 4
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'  ← 5
```

1. To use a function defined in a file you must first **import** it.
2. Call with only positional arguments
3. Call with positional arguments, one optional argument (specified by position), one optional argument (specified by name)
4. Call with positional arguments, named arguments specified by name (order is irrelevant)
5. Positional arguments are required

→

Functions summary

Functions and arguments -3

Functions summary

- Positional arguments precede named ones

Functions and arguments -3

Functions summary

- Positional arguments precede named ones
- Positional arguments are required

Functions and arguments -3

Functions summary

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

Functions and arguments -3

Functions 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 -3

Functions 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

Functions and arguments -3

Functions 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

- An empty `return` statement returns **`None`**

Functions 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

- An empty `return` statement returns **`None`**
- A function without a `return` statement terminates after the last line and returns **`None`**

Functions and arguments -3

Functions 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

- An empty `return` statement returns **`None`**
- A function without a `return` statement terminates after the last line and returns **`None`**

→

Functions and arguments -4

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

Functions and arguments -4

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)
```

Functions and arguments -4

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):  ← 1
    print("Positional required arguments (a,b):", a, b)  ← 2
    print("Standard named arguments (d):", d)  ← 3
    print("Positional variable arguments (pos):", pos)  ← 4
    print("Named variable arguments (kw):", kw)  ← 5
```

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):  ← 1
    print("Positional required arguments (a,b):", a, b)  ← 2
    print("Standard named arguments (d):", d)  ← 3
    print("Positional variable arguments (pos):", pos)  ← 4
    print("Named variable arguments (kw):", kw)  ← 5
```

1. Function **showarg()** only shows the arguments values

Functions and arguments -4

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):  ⟵ 1
    print("Positional required arguments (a,b):", a, b)  ⟵ 2
    print("Standard named arguments (d):", d)  ⟵ 3
    print("Positional variable arguments (pos):", pos)  ⟵ 4
    print("Named variable arguments (kw):", kw)  ⟵ 5
```

1. Function **showarg()** only shows the arguments values
2. **a**, **b** standard positional arguments

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):  ← 1
    print("Positional required arguments (a,b):", a, b)  ← 2
    print("Standard named arguments (d):", d)  ← 3
    print("Positional variable arguments (pos):", pos)  ← 4
    print("Named variable arguments (kw):", kw)  ← 5
```

1. Function **showarg()** only shows the arguments values
2. **a**, **b** standard positional arguments
3. **d**, standard named argument

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):  ← 1
    print("Positional required arguments (a,b):", a, b)  ← 2
    print("Standard named arguments (d):", d)  ← 3
    print("Positional variable arguments (pos):", pos)  ← 4
    print("Named variable arguments (kw):", kw)  ← 5
```

1. Function **showarg()** only shows the arguments values
2. **a**, **b** standard positional arguments
3. **d**, standard named argument
4. ***pos**, (*tuple*) variable positional arguments

Functions and arguments -4

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):          ← 1
    print("Positional required arguments (a,b):", a, b)   ← 2
    print("Standard named arguments (d):", d)    ← 3
    print("Positional variable arguments (pos):", pos)   ← 4
    print("Named variable arguments (kw):", kw)    ← 5
```

1. Function **showarg()** only shows the arguments values
2. **a**, **b** standard positional arguments
3. **d**, standard named argument
4. ***pos**, (*tuple*) variable positional arguments
5. ****kw**, (*dict*) variable named arguments

Functions and arguments -4

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):          ← 1
    print("Positional required arguments (a,b):", a, b)   ← 2
    print("Standard named arguments (d):", d)   ← 3
    print("Positional variable arguments (pos):", pos)   ← 4
    print("Named variable arguments (kw):", kw)   ← 5
```

1. Function **showarg()** only shows the arguments values
2. **a**, **b** standard positional arguments
3. **d**, standard named argument
4. ***pos**, (*tuple*) variable positional arguments
5. ****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}
```

Functions and arguments -5

Calling showarg():

```
>>> from showarg import showarg

>>> showarg(1, 2)   ← 1
Positional required arguments (a,b): 1 2
Standard named arguments (d): 15   ← 2
Positional variable arguments (pos): ()   ← 3
Named variable arguments (kw): {}   ← 3

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

Calling showarg():

```
>>> from showarg import showarg

>>> showarg(1, 2)   ← 1
Positional required arguments (a,b): 1 2
Standard named arguments (d): 15   ← 2
Positional variable arguments (pos): ()   ← 3
Named variable arguments (kw): {}   ← 3

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

1. Call with required arguments only

Calling `showarg()`:

```
>>> from showarg import showarg

>>> showarg(1, 2)   ← 1
Positional required arguments (a,b): 1 2
Standard named arguments (d): 15   ← 2
Positional variable arguments (pos): ()   ← 3
Named variable arguments (kw): {}   ← 3

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

1. Call with required arguments only
2. The optional argument gets the default value

Functions and arguments -5

Calling `showarg()`:

```
>>> from showarg import showarg

>>> showarg(1, 2)  ← 1
Positional required arguments (a,b): 1 2
Standard named arguments (d): 15  ← 2
Positional variable arguments (pos): ()  ← 3
Named variable arguments (kw): {}  ← 3

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

1. Call with required arguments only
2. The optional argument gets the default value
3. Variable arguments (not specified in the call) are empty

Calling `showarg()`:

```
>>> from showarg import showarg

>>> showarg(1, 2)                                  ← 1
Positional required arguments (a,b): 1 2
Standard named arguments (d): 15                   ← 2
Positional variable arguments (pos): ()            ← 3
Named variable arguments (kw): {}                  ← 3

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

1. Call with required arguments only
2. The optional argument gets the default value
3. Variable arguments (not specified in the call) are empty
4. Call with variable arguments both positional and named

Functions and arguments -5

Calling `showarg()`:

```
>>> from showarg import showarg

>>> showarg(1, 2)  ⟵ 1
Positional required arguments (a,b): 1 2
Standard named arguments (d): 15  ⟵ 2
Positional variable arguments (pos): ()  ⟵ 3
Named variable arguments (kw): {}  ⟵ 3

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

1. Call with required arguments only
2. The optional argument gets the default value
3. Variable arguments (not specified in the call) are empty
4. Call with variable arguments both positional and named
5. The named argument is specified positionally in the call

Functions and arguments -5

Calling showarg():

```
>>> from showarg import showarg

>>> showarg(1, 2)                                          ← 1
Positional required arguments (a,b): 1 2
Standard named arguments (d): 15                           ← 2
Positional variable arguments (pos): ()                    ← 3
Named variable arguments (kw): {}                          ← 3

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

1. Call with required arguments only
2. The optional argument gets the default value
3. Variable arguments (not specified in the call) are empty
4. Call with variable arguments both positional and named
5. The named argument is specified positionally in the call

→

Functions and arguments -6

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

Functions and arguments -6

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}
```

Functions and arguments -6

> 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)  ⟵ 1
>>> ak={"arg1":13, "arg2":14, "arg3":15}  ⟵ 2

>>> 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)  ⟵ 3
Named variable arguments (kw): {'opt1': 7, 'opt2': 8, 'arg3': 15,
 'arg2': 14, 'arg1': 13}  ⟵ 4
```

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)          ← 1
>>> ak={"arg1":13, "arg2":14, "arg3":15}   ← 2

>>> 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)   ← 3
Named variable arguments (kw): {'opt1': 7, 'opt2': 8, 'arg3': 15,
 'arg2': 14, 'arg1': 13}   ← 4
```

1. Definition of a *tuple* (**ps**) for additional positional arguments

Functions and arguments -6

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)  ⟵ 1
>>> ak={"arg1":13, "arg2":14, "arg3":15}  ⟵ 2

>>> 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) ⟵ 3
Named variable arguments (kw): {'opt1': 7, 'opt2': 8, 'arg3': 15,
 'arg2': 14, 'arg1': 13}  ⟵ 4
```

1. Definition of a *tuple* (**ps**) for additional positional arguments
2. Definition of a *dictionary* (**ak**) for additional named arguments

Functions and arguments -6

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)  ⟵ 1
>>> ak={"arg1":13, "arg2":14, "arg3":15}  ⟵ 2

>>> 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)  ⟵ 3
Named variable arguments (kw): {'opt1': 7, 'opt2': 8, 'arg3': 15,
 'arg2': 14, 'arg1': 13}  ⟵ 4
```

1. Definition of a *tuple* (**ps**) for additional positional arguments
2. Definition of a *dictionary* (**ak**) for additional named arguments
3. The **pos** *tuple* in the function “receives” positional parameters other than required ones and the content of **ps** *tuple* from the call

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)          ← 1
>>> ak={"arg1":13, "arg2":14, "arg3":15}          ← 2

>>> 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)          ← 3
Named variable arguments (kw): {'opt1': 7, 'opt2': 8, 'arg3': 15,
 'arg2': 14, 'arg1': 13}          ← 4
```

1. Definition of a *tuple* (**ps**) for additional positional arguments
2. Definition of a *dictionary* (**ak**) for additional named arguments
3. The **pos** *tuple* in the function “receives” positional parameters other than required ones and the content of **ps** *tuple* from the call
4. The **kw** *dictionary* in the function “receives” named variable arguments and the content of **ak** *dictionary* from the call

Functions and arguments -6

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)                                ← 1
>>> ak={"arg1":13, "arg2":14, "arg3":15}         ← 2

>>> 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) ← 3
Named variable arguments (kw): {'opt1': 7, 'opt2': 8, 'arg3': 15,
 'arg2': 14, 'arg1': 13}                          ← 4
```

1. Definition of a *tuple* (**ps**) for additional positional arguments
2. Definition of a *dictionary* (**ak**) for additional named arguments
3. The **pos** *tuple* in the function “receives” positional parameters other than required ones and the content of **ps** *tuple* from the call
4. The **kw** *dictionary* in the function “receives” named variable arguments and the content of **ak** *dictionary* from the call

→

End of Part I