# Computational Graphics: Lecture 2

Francesco Furiani
(fra.uni@furio.me)

Università degli Studi Roma TRE

Tue, Mar 4, 2014

# Why Python?

Well that's what the prof. decided :)
On a more serious note:

- Interpreted and interactive language

# Why Python?

Well that's what the prof. decided :)
On a more serious note:

- Interpreted and interactive language
- Strong dynamic typing system

# Why Python?

Well that's what the prof. decided :)
On a more serious note:

- Interpreted and interactive language
- Strong dynamic typing system
- Multiple programming paradigms

# Why Python?

Well that's what the prof. decided :)
On a more serious note:

- Interpreted and interactive language
- Strong dynamic typing system
- Multiple programming paradigms
- Cross-platform

# Why Python?

Well that's what the prof. decided :)
On a more serious note:

- Interpreted and interactive language
- Strong dynamic typing system
- Multiple programming paradigms
- Cross-platform
- Thought for begineers ...

# Why Python?

Well that's what the prof. decided :)
On a more serious note:

- Interpreted and interactive language
- Strong dynamic typing system
- Multiple programming paradigms
- Cross-platform
- Thought for begineers ...
- ... and computer scientist

# Why Python?

Well that's what the prof. decided :)
On a more serious note:

- Interpreted and interactive language
- Strong dynamic typing system
- Multiple programming paradigms
- Cross-platform
- Thought for begineers ...
- ... and computer scientist
- Easy to read

# Why Python?

Well that's what the prof. decided :)
On a more serious note:

- Interpreted and interactive language
- Strong dynamic typing system
- Multiple programming paradigms
- Cross-platform
- Thought for begineers ...
- ... and computer scientist
- Easy to read
- Large and comprehensive standard library

# History

From "Python mailing list"

> *>Why the name "python"?*
>
> *I an a fan of Monty Python's Flying Circus, and I hate acronyms. Other than that, there's nothing deep.*
>
> *>What role do you see python as fulfilling or in what direction do you anticipate its evolution?*
>
> *The one thing that Python definitely does not want to be is a GENERAL purpose programming language. Its lack of declarations and general laziness about compile-time checking is definitely aimed at small-to-medium-sized programs. [...]*

Guido van Rossum is now working at Dropbox after being employed by Google, DARPA and NIST.

# Install steps

Windows

32bit http://www.python.org/ftp/python/2.7.7/python-2.7.7.msi

Linux

MacOSX

# Install steps

Windows

32bit http://www.python.org/ftp/python/2.7.7/python-2.7.7.msi

64bit http://www.python.org/ftp/python/2.7.7/python-2.7.7.msi

Linux

MacOSX

# Install steps

Windows

32bit http://www.python.org/ftp/python/2.7.7/python-2.7.7.msi

64bit http://www.python.org/ftp/python/2.7.7/python-2.7.7.msi

Linux

Already present in most of the distributions, if your is one of the few missing it, just use the package manager to install it.

MacOSX

# Install steps

Windows

32bit http://www.python.org/ftp/python/2.7.7/python-2.7.7.msi

64bit http://www.python.org/ftp/python/2.7.7/python-2.7.7.msi

Linux

Already present in most of the distributions, if your is one of the few missing it, just use the package manager to install it.

MacOSX

Like Linux is a part of the OS, optionally you can install XCode for more tools.

# How to use

Open the shell and run the python cli command (`python`) or, if you're using Windows, fire up the *Python interactive shell*.

## Result

```
furio@suppaman:~$ python
Python 2.7.3 (default, Sep 26 2013, 20:03:06)
[GCC 4.6.3] on linux2
>>>
```

## Exiting the interpreter

```
>>> exit()
furio@suppaman:~$
```

# How to use

Create a py file and put your code inside. Execute with `python filename.py`

## Example with shell

```
furio@suppaman:~$ echo "print('Un programma python')" > esempio.py
furio@suppaman:~$ python esempio.py
Un programma python
```

# Indent style

Indentation is used by Python interpreter to determine in which code block is executing, so you need to take care of proper tabulations!

```
def fact(x):
  if x == 0:
    return 1
  else:
    return x * fact(x-1)
```

Always use an editor that treats tabulation characters with care, like Sublime Text (free, cross-platform).

# Types

Python is a *strongly* typed language.

```
>>> a = 2
>>> type(a)
<type 'int'>

>>> a = 2L
>>> type(a)
<type 'long'>

>>> a = False
>>> type(a)
<type 'bool'>

>>> a = "Ciao"
>>> type(a)
<type 'str'>

>>> a = 1.
>>> type(a)
<type 'float'>
```

# Types

Python is a *strongly* typed language.

```
>>> a = 2
>>> type(a)
<type 'int'>

>>> a = 2L
>>> type(a)
<type 'long'>

>>> a = False
>>> type(a)
<type 'bool'>

>>> a = "Ciao"
>>> type(a)
<type 'str'>

>>> a = 1.
>>> type(a)
<type 'float'>
```

```
>>> a = []
>>> type(a)
<type 'list'>

>>> a = ()
>>> type(a)
<type 'tuple'>

>>> a = {}
>>> type(a)
<type 'dict'>

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

>>> import numpy
>>> a = numpy.ndarray((1,1))
>>> type(a)
<type 'numpy.ndarray'>
```

# Types

Python is a *strongly* typed language.

```
>>> a = 2
>>> type(a)
<type 'int'>

>>> a = 2L
>>> type(a)
<type 'long'>

>>> a = False
>>> type(a)
<type 'bool'>

>>> a = "Ciao"
>>> type(a)
<type 'str'>

>>> a = 1.
>>> type(a)
<type 'float'>
```

```
>>> a = []
>>> type(a)
<type 'list'>

>>> a = ()
>>> type(a)
<type 'tuple'>

>>> a = {}
>>> type(a)
<type 'dict'>

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

>>> import numpy
>>> a = numpy.ndarray((1,1))
>>> type(a)
<type 'numpy.ndarray'>
```

# Operators

```
 ** # Exponentiation (raise to the power)
~ + - # Complement, unary plus and minus (method names for the last
      two are +@ and -@)
* / % // # Multiply, divide, modulo and floor division
+ - # Addition and subtraction
>> << # Right and left bitwise shift
& # Bitwise 'AND'
^ | # Bitwise exclusive 'OR' and regular 'OR'
<= < > >= # Comparison operators
<> == != # Equality operators
= %= /= //= -= += *= **= # Assignment operators
is is not # Identity operators
in not in # Membership operators
not or and # Logical operators
```

# If, Else, While, Def

```
def magicsort( aList ):
  _magicsort( aList, 0, len( aList ) - 1 )

def _magicsort( aList, first, last ):
  mid = ( first + last ) / 2
  if first < last:
    _magicsort( aList, first, mid )
    _magicsort( aList, mid + 1, last )

  a, f, l = 0, first, mid + 1
  tmp = [None] * ( last - first + 1 )

  while f <= mid and l <= last:
    if aList[f] < aList[l] :
      tmp[a] = aList[f]
      f += 1
    else:
      tmp[a] = aList[l]
      l += 1
    a += 1
```

# If, Else, While, Def

```
def magicsort( aList ):
  _magicsort( aList, 0, len( aList ) - 1 )

def _magicsort( aList, first, last ):
  mid = ( first + last ) / 2
  if first < last:
    _magicsort( aList, first, mid )
    _magicsort( aList, mid + 1, last )

  a, f, l = 0, first, mid + 1
  tmp = [None] * ( last - first + 1 )

  while f <= mid and l <= last:
    if aList[f] < aList[l] :
      tmp[a] = aList[f]
      f += 1
    else:
      tmp[a] = aList[l]
      l += 1
    a += 1
```

```
if f <= mid :
    tmp[a:] = aList[f:mid + 1]

  if l <= last:
    tmp[a:] = aList[l:last + 1]

  a = 0
  while first <= last:
    aList[first] = tmp[a]
    first += 1
    a += 1
```

What this code does? Super prize for the one that answer! (Hint it's a sort you know already)

# If, Else, While, Def

```
def magicsort( aList ):
  _magicsort( aList, 0, len( aList ) - 1 )

def _magicsort( aList, first, last ):
  mid = ( first + last ) / 2
  if first < last:
    _magicsort( aList, first, mid )
    _magicsort( aList, mid + 1, last )

  a, f, l = 0, first, mid + 1
  tmp = [None] * ( last - first + 1 )

  while f <= mid and l <= last:
    if aList[f] < aList[l] :
      tmp[a] = aList[f]
      f += 1
    else:
      tmp[a] = aList[l]
      l += 1
    a += 1
```

```
if f <= mid :
    tmp[a:] = aList[f:mid + 1]

  if l <= last:
    tmp[a:] = aList[l:last + 1]

  a = 0
  while first <= last:
    aList[first] = tmp[a]
    first += 1
    a += 1
```

What this code does? Super prize for the one that answer! (Hint it's a sort you know already)

# Function definition

A function is a code segment that is callable and owns a scope in which executes.

```
def fact(x):
  if x == 0:
    return 1
  else:
    return x * fact(x-1)
```

You need to define it with the keyword `def` (ah!) followed by the function name and an optional list of arguments

```
def func(x,y,z):
  print '{0} {1} {2}'.format(x,y,z)

def func(x,y,z=2): # If not specified 'z' has value 2
  print '{0} {1} {2}'.format(x,y,z)
```

# Function definition

Calling a function is similar as other languages you might have used.

```
>>> func(1,10)
1 10 2
>>> func(1,10,7)
1 10 7
```

But it can support nominal parameters call!

```
>>> func(y=2,z=5,x=4)
4 2 5
```

...

# Function definition

...

Or an unknown list of parameters

```
>>> def func(*a): print [arg for arg in a]
>>> func(1,2,'ciccio')
[1, 2, 'ciccio']
```

Or an unknown list of named parameters

```
>>> def func(**a): print [(name,value) for name,value in a.items()]
>>> func(a=1,b=2,c='ciccio')
[('a', 1), ('c', 'ciccio'), ('b', 2)]
```

# Lists - 1

## Definition

```
>>> x = []
>>> y = ['Ciao', 'ragazzi']
>>> z = ['Numeri', 1, 'a', 22222,
     'caso']

>>> len(y)
2

>>> for i in y: print i
'Ciao'
'ragazzi'

>>> del y[0]
>>> y
['ragazzi']

>>> k = [[1],[2],[3]]
```

# Lists - 1

## Definition

```
>>> x = []
>>> y = ['Ciao', 'ragazzi']
>>> z = ['Numeri', 1, 'a', 22222,
     'caso']

>>> len(y)
2

>>> for i in y: print i
'Ciao'
'ragazzi'

>>> del y[0]
>>> y
['ragazzi']

>>> k = [[1],[2],[3]]
```

## Unary operators

```
>>> [1] + [2]
[1, 2]

>>> [4] * 4
[4, 4, 4, 4]

>>> 3 in [3,2,1]
True

>>> [1,2,3] == [1,2,3]
True

>>> [1,2,3] > [2,3,1]
False

>>> [1,2,3] < [2,3,1]
True
```

# Lists - 1

## Definition

```
>>> x = []
>>> y = ['Ciao', 'ragazzi']
>>> z = ['Numeri', 1, 'a', 22222,
     'caso']

>>> len(y)
2

>>> for i in y: print i
'Ciao'
'ragazzi'

>>> del y[0]
>>> y
['ragazzi']

>>> k = [[1],[2],[3]]
```

## Unary operators

```
>>> [1] + [2]
[1, 2]

>>> [4] * 4
[4, 4, 4, 4]

>>> 3 in [3,2,1]
True

>>> [1,2,3] == [1,2,3]
True

>>> [1,2,3] > [2,3,1]
False

>>> [1,2,3] < [2,3,1]
True
```

# Lists - 10

## Address / Slicing

```
>>> L
[1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> L[2]
3
>>> L[-2]
8
>>> L[7-len(L)]
8

>>> L[1:]
[2, 3, 4, 5, 6, 7, 8, 9]
>>> L
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> L[1:5]
[2, 3, 4, 5]
>>> L[:5]
[1, 2, 3, 4, 5]
```

# Lists - 10

## Address / Slicing

```
>>> L
[1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> L[2]
3
>>> L[-2]
8
>>> L[7-len(L)]
8

>>> L[1:]
[2, 3, 4, 5, 6, 7, 8, 9]
>>> L
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> L[1:5]
[2, 3, 4, 5]
>>> L[:5]
[1, 2, 3, 4, 5]
```

## Java-like functions

```
list.append(obj)

list.count(obj)

list.extend(seq)

list.index(obj)

list.insert(index, obj)

list.pop(obj=list[-1])

list.remove(obj)

list.reverse()

list.sort([func])
```

# Lists - 10

## Address / Slicing

```
>>> L
[1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> L[2]
3
>>> L[-2]
8
>>> L[7-len(L)]
8

>>> L[1:]
[2, 3, 4, 5, 6, 7, 8, 9]
>>> L
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> L[1:5]
[2, 3, 4, 5]
>>> L[:5]
[1, 2, 3, 4, 5]
```

## Java-like functions

```
list.append(obj)

list.count(obj)

list.extend(seq)

list.index(obj)

list.insert(index, obj)

list.pop(obj=list[-1])

list.remove(obj)

list.reverse()

list.sort([func])
```

# Lists - 11

## Creation

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(0,10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(0,10,2)
[0, 2, 4, 6, 8]
>>> range(10,0,-2)
[10, 8, 6, 4, 2]
```

What if we need to build a billion elements long list?

```
>>> range(1000000000)
```

NO!!!

```
>>> xrange(1000000000)
```

# Lists - 11

## Creation

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(0,10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(0,10,2)
[0, 2, 4, 6, 8]
>>> range(10,0,-2)
[10, 8, 6, 4, 2]
```

What if we need to build a billion elements long list?

```
>>> range(1000000000)
```

NO!!!

```
>>> xrange(1000000000)
```

## Generators

When necessary it returns the element!
It might be useful (for debug purpose) to make the generator expand all the elements:

```
>>> list(xrange(1000000000))
```

It can be easily conceived:

```
>>> from itertools import count
>>> S = (2*x for x in count() if
    x**2 > 3)
```

.. that goes on and on and on ...

# Lists - 11

## Creation

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(0,10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(0,10,2)
[0, 2, 4, 6, 8]
>>> range(10,0,-2)
[10, 8, 6, 4, 2]
```

What if we need to build a billion elements long list?

```
>>> range(1000000000)
```

NO!!!

```
>>> xrange(1000000000)
```

## Generators

When necessary it returns the element!
It might be useful (for debug purpose) to make the generator expand all the elements:

```
>>> list(xrange(1000000000))
```

It can be easily conceived:

```
>>> from itertools import count
>>> S = (2*x for x in count() if
    x**2 > 3)
```

.. that goes on and on and on ...

# Lists - 100

## Comprehension

```
>>> [x*x for x in xrange(1,11)]
[1, 4, 9, 16, 25, 36, 49, 64, 81,
     100]
>>> [x*x for x in range(10) if x
    % 2 == 0]
[0, 4, 16, 36, 64]

>>> x = 3
>>> [[int(i==j) for i in range(x)
    ] for j in range(x)]
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]

>>> [a+str(b) for a in 'ABC' for
    b in xrange(1,4)]
['A1', 'A2', 'A3', 'B1', 'B2', '
    B3', 'C1', 'C2', 'C3']
```

# Lists - 100

## Comprehension

```
>>> [x*x for x in xrange(1,11)]
[1, 4, 9, 16, 25, 36, 49, 64, 81,
     100]
>>> [x*x for x in range(10) if x
    % 2 == 0]
[0, 4, 16, 36, 64]

>>> x = 3
>>> [[int(i==j) for i in range(x)
    ] for j in range(x)]
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]

>>> [a+str(b) for a in 'ABC' for
    b in xrange(1,4)]
['A1', 'A2', 'A3', 'B1', 'B2', '
    B3', 'C1', 'C2', 'C3']
```

## Condtional expression

```
>>> x = 3
>>> [[1 if j == i else 0 for j in
     range(x)] for i in range(x)]
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]

>>> filter(lambda y: y != -1, [x*
    x if x%2 == 0 else -1 for x
    in xrange(10)])
[0, 4, 16, 36, 64]
```

Filter? Lambda?

# Lists - 100

## Comprehension

```
>>> [x*x for x in xrange(1,11)]
[1, 4, 9, 16, 25, 36, 49, 64, 81,
     100]
>>> [x*x for x in range(10) if x
    % 2 == 0]
[0, 4, 16, 36, 64]

>>> x = 3
>>> [[int(i==j) for i in range(x)
    ] for j in range(x)]
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]

>>> [a+str(b) for a in 'ABC' for
    b in xrange(1,4)]
['A1', 'A2', 'A3', 'B1', 'B2', '
    B3', 'C1', 'C2', 'C3']
```

## Condtional expression

```
>>> x = 3
>>> [[1 if j == i else 0 for j in
     range(x)] for i in range(x)]
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]

>>> filter(lambda y: y != -1, [x*
    x if x%2 == 0 else -1 for x
    in xrange(10)])
[0, 4, 16, 36, 64]
```

Filter? Lambda?

# Lists - 101

## Manipolation

- `filter(func, list)` Filter the list elements with the `func` predicate.
- `map(func, list[, list, ])` Execute the `func` predicate on the input list(s).

# Lists - 101

## Manipolation

- `filter(func, list)` Filter the list elements with the `func` predicate.
- `map(func, list[, list, ])` Execute the `func` predicate on the input list(s).

```
>>> def isEven(x): return x % 2
    == 0
>>> filter(isEven, range(10))
[0, 2, 4, 6, 8]
>>> filter(lambda x: x % 2 == 0,
    range(10))
[0, 2, 4, 6, 8]
```

```
>>> def mulSelf(x): return x*x
>>> map(mulSelf, range(10))
[0, 1, 4, 9, 16, 25, 36, 49, 64,
    81]
>>> map(lambda x: x * x, range
    (10))
[0, 1, 4, 9, 16, 25, 36, 49, 64,
    81]
```

# Lists - 101

## Manipolation

- `filter(func, list)` Filter the list elements with the `func` predicate.
- `map(func, list[, list, ])` Execute the `func` predicate on the input list(s).

```
>>> def isEven(x): return x % 2
    == 0
>>> filter(isEven, range(10))
[0, 2, 4, 6, 8]
>>> filter(lambda x: x % 2 == 0,
    range(10))
[0, 2, 4, 6, 8]
```

```
>>> def mulSelf(x): return x*x
>>> map(mulSelf, range(10))
[0, 1, 4, 9, 16, 25, 36, 49, 64,
    81]
>>> map(lambda x: x * x, range
    (10))
[0, 1, 4, 9, 16, 25, 36, 49, 64,
    81]
```

# Lists - 110

## Manipulation

- `reduce(func, list[, init])` Apply the function `func` on the list elements left-to-right returning the accumulated result created by the `func` execution.
- `zip(list[, list, ])` Combine the elements of input lists stopping when the shortest input list ends.

# Lists - 110

## Manipulation

- `reduce(func, list[, init])` Apply the function `func` on the list elements left-to-right returning the accumulated result created by the `func` execution.
- `zip(list[, list, ])` Combine the elements of input lists stopping when the shortest input list ends.

```
>>> def sumReduce(x,y): return x+
    y
>>> reduce(sumReduce, range(10))
45
>>> reduce(lambda x,y: x+y, range
    (10))
45
>>> sum(range(10))
45
```

```
>>> zip(range(1,4),range(5,9))
[(1, 5), (2, 6), (3, 7)]
```

# Lists - 110

## Manipulation

- `reduce(func, list[, init])` Apply the function `func` on the list elements left-to-right returning the accumulated result created by the `func` execution.
- `zip(list[, list, ])` Combine the elements of input lists stopping when the shortest input list ends.

```
>>> def sumReduce(x,y): return x+
    y
>>> reduce(sumReduce, range(10))
45
>>> reduce(lambda x,y: x+y, range
    (10))
45
>>> sum(range(10))
45
```

```
>>> zip(range(1,4),range(5,9))
[(1, 5), (2, 6), (3, 7)]
```

# Tuple — An immutable list :)

Once initialized it'll stay the same till the interpreter death ( :o ) and/or the variable deletion / collection.

Is possible to use the same things we learned about lists on tuples: they'll either work returning a copy of the tuple as a list or throw an error :)

```
>>> a = (1,2,3)
>>> a[0]
1
>>> del a[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item deletion
>>> map(lambda x: x+1, a)
[2, 3, 4]
```

# Dictionary

Dictionaries are similar to other composite types but they allow to use, as index, any immutable type.

```
>>> Eng2Ita = {}
>>> Eng2Ita['one'] = 'uno'
>>> Eng2Ita['two'] = 'due'
>>> print Eng2Ita['two']
'due'
```

# Dictionary

Dictionaries are similar to other composite types but they allow to use, as index, any immutable type.

```
>>> Eng2Ita = {}
>>> Eng2Ita['one'] = 'uno'
>>> Eng2Ita['two'] = 'due'
>>> print Eng2Ita['two']
'due'
```

## Access methods

```
>>> Eng2Ita = {'one':'uno', 'two'
    :'due', 'three':'tre'}

>>> Eng2Ita.keys()
['one', 'three', 'two']
>>> Eng2Ita.values()
['uno', 'tre', 'due']
>>> Eng2Ita.items()
[('one','uno'), ('three', 'tre'),
     ('two', 'due')]

>>> Eng2Ita.has_key('one')
1
```

# Dictionary

Dictionaries are similar to other composite types but they allow to use, as index, any immutable type.

```
>>> Eng2Ita = {}
>>> Eng2Ita['one'] = 'uno'
>>> Eng2Ita['two'] = 'due'
>>> print Eng2Ita['two']
'due'
```

## Access methods

```
>>> Eng2Ita = {'one':'uno', 'two'
    :'due', 'three':'tre'}

>>> Eng2Ita.keys()
['one', 'three', 'two']
>>> Eng2Ita.values()
['uno', 'tre', 'due']
>>> Eng2Ita.items()
[('one','uno'), ('three', 'tre'),
     ('two', 'due')]

>>> Eng2Ita.has_key('one')
1
```

# Dictionary

A dictionary, for example, can be used to represent a sparse matrix.

```
>>> Matrice = {(0,3): 1, (2, 1):
    2, (4, 3): 3}
```

We can use tuples as keys (in which we store the $i, j$ indexes) and we store only the non zero values.
To access an element we can use the [] operator.

# Dictionary

A dictionary, for example, can be used to represent a sparse matrix.

```
>>> Matrice = {(0,3): 1, (2, 1):
    2, (4, 3): 3}
```

We can use tuples as keys (in which we store the $i, j$ indexes) and we store only the non zero values.
To access an element we can use the [] operator.

```
>>> Matrice[(0,3)]
1
>>> Matrice[0,3] # equivalent
1
```

The [] operator returns an error if the key is not present, to avoid that, we can use the get operator that offers a default value to return if the key is not present in the dictionary:

```
>>> Matrice[1,3]
KeyError: (1, 3)
>>> Matrice.get((1,3), 0)
0
```

# Dictionary

A dictionary, for example, can be used to represent a sparse matrix.

```
>>> Matrice = {(0,3): 1, (2, 1):
    2, (4, 3): 3}
```

We can use tuples as keys (in which we store the $i, j$ indexes) and we store only the non zero values.
To access an element we can use the [] operator.

```
>>> Matrice[(0,3)]
1
>>> Matrice[0,3] # equivalent
1
```

The [] operator returns an error if the key is not present, to avoid that, we can use the get operator that offers a default value to return if the key is not present in the dictionary:

```
>>> Matrice[1,3]
KeyError: (1, 3)
>>> Matrice.get((1,3), 0)
0
```

# References

Official

Site

Books

Tutorial

# References

Official

Site

Docs

Books

Tutorial

# References

Official

Site

Docs

Books

Think Python (FREE)

Tutorial

# References

Official

- Site
- Docs

Books

- Think Python (FREE)
- Python in a Nutshell

Tutorial

# References

Official

- Site
- Docs

Books

- Think Python (FREE)
- Python in a Nutshell

Tutorial

- Quick tutorial

# References

Official

- Site
- Docs

Books

- Think Python (FREE)
- Python in a Nutshell

Tutorial

- Quick tutorial
- Learn Python the hard way


On the next lecture: more Python cat!