Python

BASICS

Introduction to Python programming, basic concepts: formatting, naming conventions, variables, etc.

POLITECNICO DI TORINO

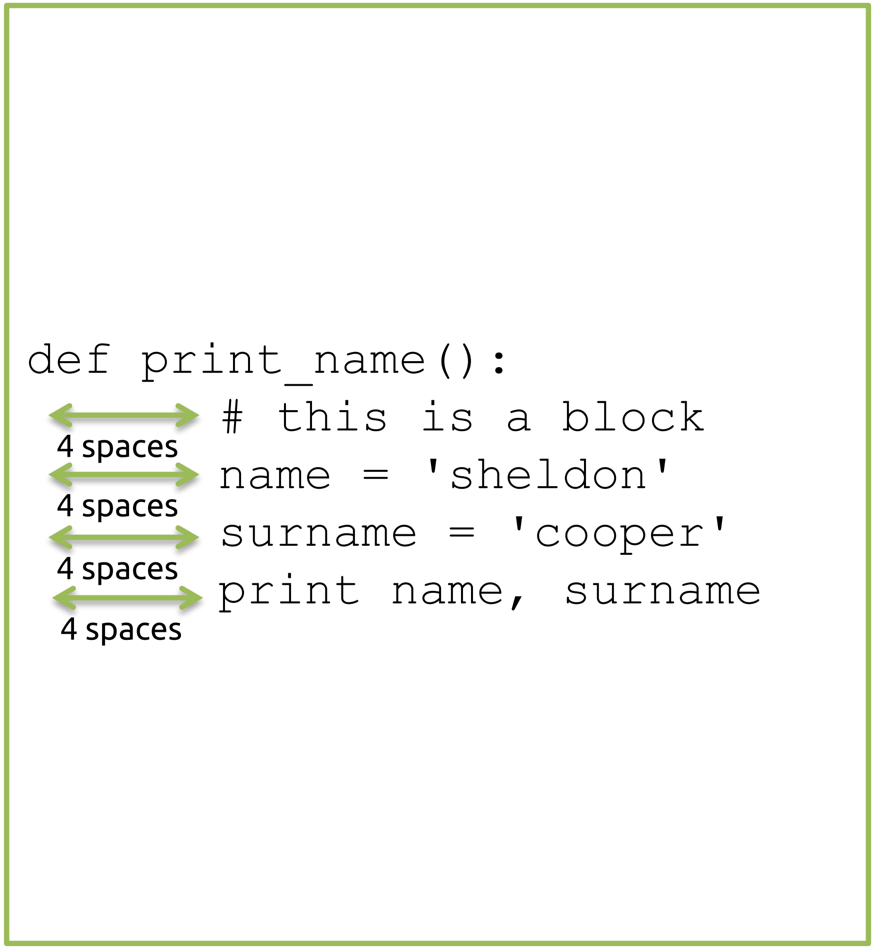
Editing / Formatting

- Python programs are text files
- The end of a line marks the end of a statement
- Comments:
  - Inline comments start with a #

```
print 1+1 #statement

# inline comment
```

Editing / Formatting

- Code blocks are defined through identation
  - mandatory
  - 4 spaces strategy
    - Use 4 spaces for code identation
    - Configure the text editor to replace tabs with 4 spaces (default in PyDev)
    - Exploit automatic identation



```
def print_name():
    # this is a block
    name = 'sheldon'
    surname = 'cooper'
    print name, surname
```

Keywords

- and
- del
- from
- not
- while
- as
- elif
- global
- or
- with

- assert
- else
- if
- pass
- yield
- break
- except
- import
- print
- class
- exec

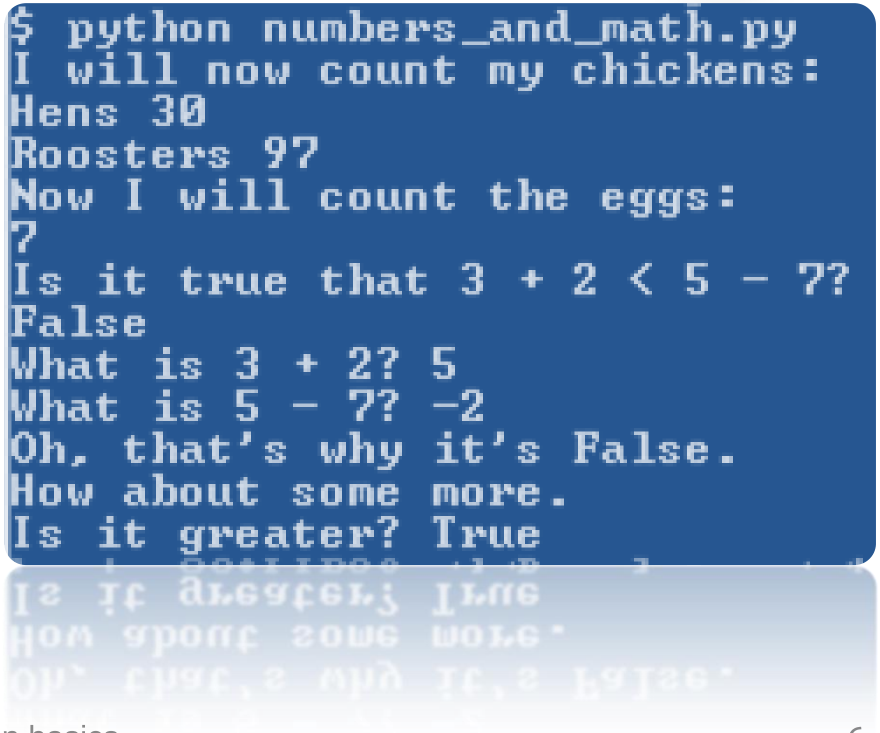
- in
- raise
- continue
- finally
- is
- return
- def
- for
- lambda
- try

Numbers and math

Operator	Description
+ plus	Sum
- minus	Subtraction
/ slash	Floor division
* asterisk	Multiplication
** double asterisk	Exponentiation
% percent	Remainder
< less-than	Comparison
> greater-than	Comparison
<= less-than-equal	Comparison
>= greater-than-equal	Comparison

Numbers and math

```
print "I will now count my chickens:"
print "Hens", 25 + 30 / 6
print "Roosters", 100 - 25 * 3 % 4
print "Now I will count the eggs:"
print 3 + 2 + 1 - 5 + 4 % 2 - 1 / 4 + 6
print "Is it true that 3 + 2 < 5 - 7?"
print 3 + 2 < 5 - 7
print "What is 3 + 2?", 3 + 2
print "What is 5 - 7?", 5 - 7
print "Oh, that's why it's False."
print "How about some more."
print "Is it greater?", 5 > -2
```



```
$ python numbers_and_math.py
I will now count my chickens:
Hens 30
Roosters 97
Now I will count the eggs:
7
Is it true that 3 + 2 < 5 - 7?
False
What is 3 + 2? 5
What is 5 - 7? -2
Oh, that's why it's False.
How about some more.
Is it greater? True
```

Order of operations

- PEMDAS
  - Parenthesis
  - Exponentiation
  - Multiplication
  - Division
  - Addition
  - Subtraction
- Same precedence
  - Left to right execution

Naming conventions

- `joined_lower`
  - for functions, variables, attributes
- `joined_lower` or `ALL_CAPS`
  - for constants
- `StudlyCaps`
  - for classes

```
#variables
my_variable = 12
my_second_variable = 'Hello!'

#functions
my_function(my_variable)
my_print(my_second_variable)
```

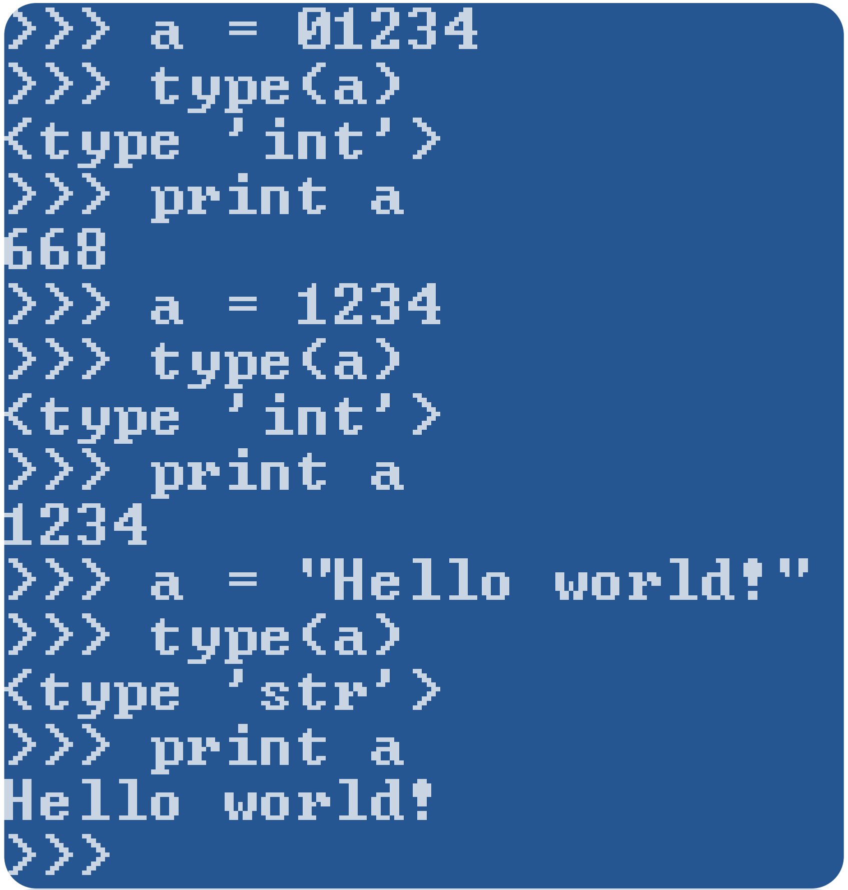
Variables

- Variable types are not explicitly declared
- Runtime type-checking
- The same variable can be reused for holding different data types

```
#integer variable
a = 1
print a

#float variable
a = 2.345
print a

#re-assignment to string
a = 'my name'
print a

# double quotes could be
# used as well
a = "my name"
print a
```

More variables

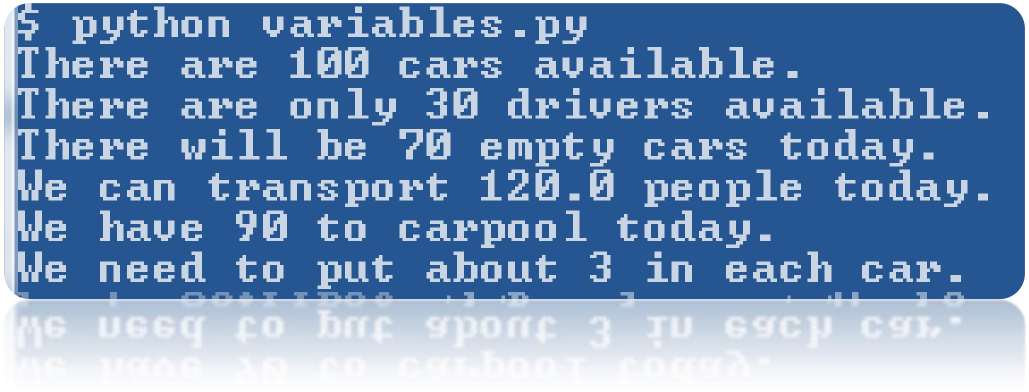
- Actual type can be checked through the interpreter
- Check the first result, what happened?
  - Display 01,010,01010
  - Display 08
  - Octal numbering system?



```
>>> a = 01234
>>> type(a)
<type 'int'>
>>> print a
668
>>> a = 1234
>>> type(a)
<type 'int'>
>>> print a
1234
>>> a = "Hello world!"
>>> type(a)
<type 'str'>
>>> print a
Hello world!
>>>
```

Examples

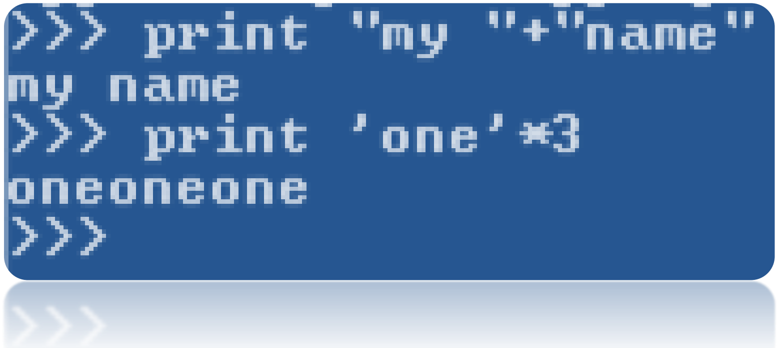
```
cars = 100
space_in_a_car = 4.0
drivers = 30
passengers = 90
cars_not_driven = cars - drivers
cars_driven = drivers
carpool_capacity = cars_driven * space_in_a_car
average_passengers_per_car = passengers / cars_driven

print 'There are', cars, 'cars available.'
print 'There are only', drivers, 'drivers available.'
print 'There will be', cars_not_driven, 'empty cars today.'
print 'We can transport', carpool_capacity, 'people today.'
print 'We have', passengers, 'to carpool today.'
print 'We need to put about', average_passengers_per_car,'in each car.'
```



```
$ python variables.py
There are 100 cars available.
There are only 30 drivers available.
There will be 70 empty cars today.
We can transport 120.0 people today.
We have 90 to carpool today.
We need to put about 3 in each car.
```

Strings

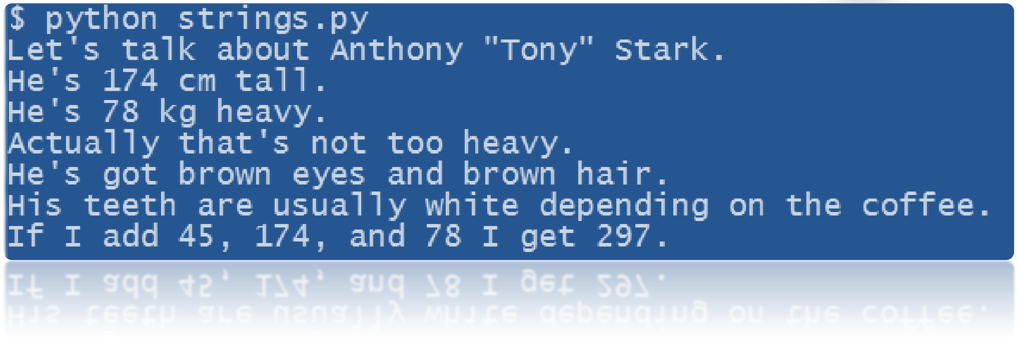
- Defined by using quotes
  - "first string"
  - 'second string'
- Immutable
- Each character in a string is assigned a number
  - the number is called *index*
- Mathematical operators cannot be applied
- Exceptions
  - + : means concatenation
  - * : means repetition



```
>>> print "my "+"name"
my name
>>> print 'one'*3
oneoneone
>>>
```

Strings

```
name = 'Anthony "Tony" Stark'
age = 45 # not a lie
height = 174 # cm
weight = 78 # kg
eyes = 'brown'
teeth = 'white'
hair = 'brown'
```



```
$ python strings.py
Let's talk about Anthony "Tony" Stark.
He's 174 cm tall.
He's 78 kg heavy.
Actually that's not too heavy.
He's got brown eyes and brown hair.
His teeth are usually white depending on the coffee.
If I add 45, 174, and 78 I get 297.
```

```
print "Let's talk about %s." % name
print "He's %d cm tall." % height
print "He's %d pounds heavy." % weight
print "Actually that's not too heavy."
print "He's got %s eyes and %s hair." % (eyes, hair)
print "His teeth are usually %s depending on the coffee." % teeth
# this line is tricky, try to get it exactly right
print "If I add %d, %d, and %d I get %d." % (age, height, weight, age + height + weight)
```

Strings

Specifiers

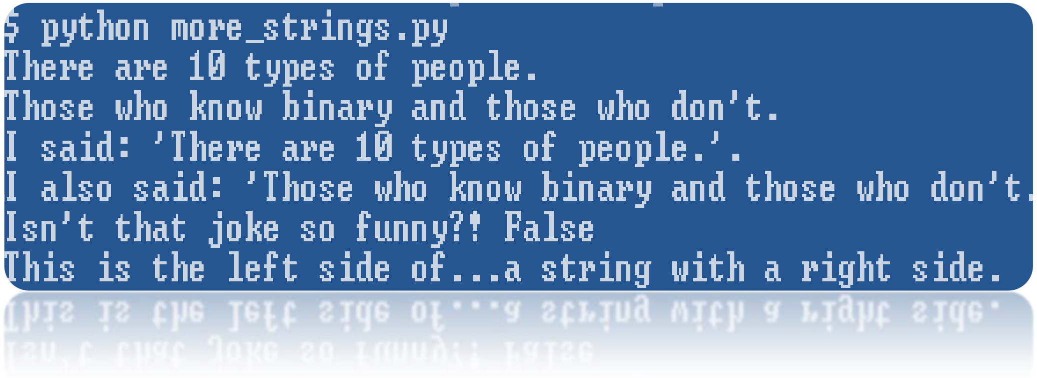
- %s, format strings
- %d, format numbers
- %r, raw representation

```
name = 'Antho
age = 45 # not
height = 174 #
weight = 78 # kg
eyes = 'brown'
teeth = 'white'
hair = 'brown'

print "Let's talk about %s." % name
print "He's %d cm tall." % height
print "He's %d pounds heavy." % weight
print "Actually that's not too heavy."
print "He's got %s eyes and %s hair." % (eyes, hair)
print "His teeth are usually %s depending on the coffee." % teeth
# this line is tricky, try to get it exactly right
print "If I add %d, %d, and %d I get %d." % (age, height, weight, age + height + weight)
```

Tuple

More strings

```
x = "There are %d types of people." % 10
binary = "binary"
do_not = "don't"
y = "Those who know %s and those who %s." % (binary, do_not)
print x
print y
print "I said: %r." % x
print "I also said: '%s'." % y
hilarious = False
joke_evaluation = "Isn't that joke so funny?! %r"
print joke_evaluation % hilarious
w = "This is the left side of..."
e = "a string with a right side."
print w + e
```



```
$ python more_strings.py
There are 10 types of people.
Those who know binary and those who don't.
I said: 'There are 10 types of people.'.
I also said: 'Those who know binary and those who don't.
Isn't that joke so funny?! False
This is the left side of...a string with a right side.
```

Escape sequences

- \n
  - Line feed + Carriage return
- \\
  - Prints a «\»
- We want to print «Hello»
  - **print** "I said: "Hello" "
  - Syntax error: no difference between quotes
- Solution: using escape sequences
  - **print** "I said: \“Hello\" "

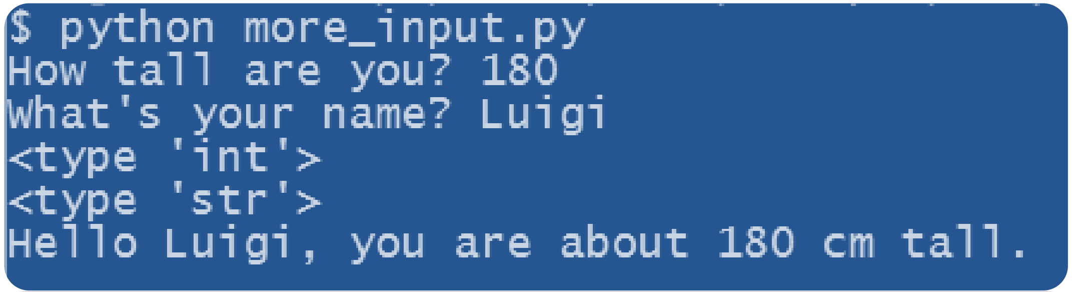
Getting input from people

- Asking questions
  - We want to ask the user’s age
  - We want to ask the user’s height
- The `raw_input()` function allows to read from the console

```
print "How old are you?",
age = raw_input()
print "How tall are you?",
height = raw_input()
print "You are %s years old, and you are about %s cm tall." % (age, height)
```

More input

```
height = int(raw_input("How tall are you? "))
name = raw_input("What's your name? ")
print type(height)
print type(name)

print("Hello %s, you are about %d tall" %(name, height) )
```



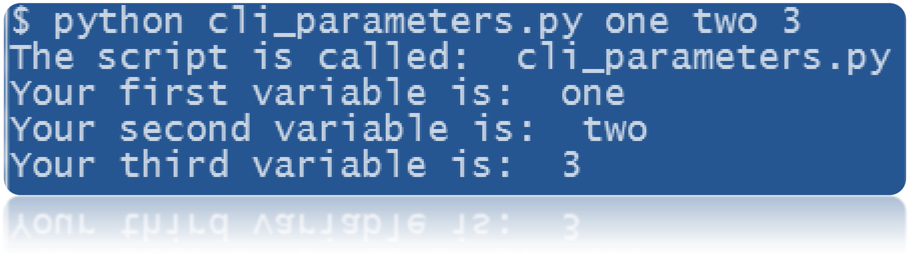
```
$ python more_input.py
How tall are you? 180
What's your name? Luigi
<type 'int'>
<type 'str'>
Hello Luigi, you are about 180 cm tall.
```

Command-line parameters

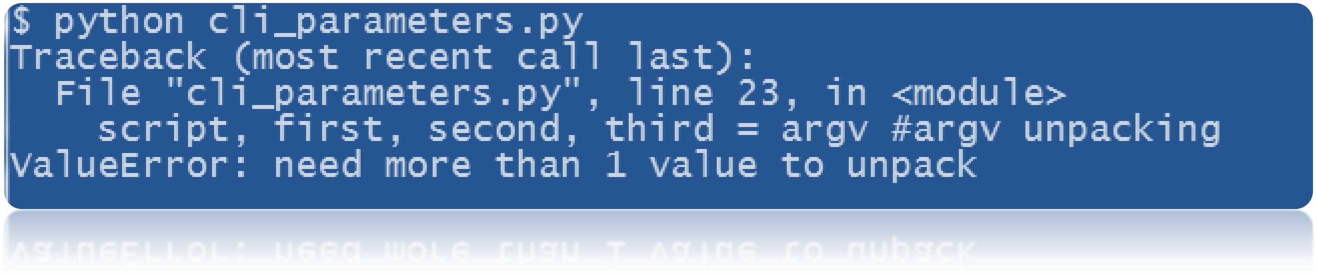
- Python scripts can receive launch parameters
  - Placed just after the script name
  - Any number
  - Accessible through sys.argv
- sys
  - Python module to handle system-level operations
- argv
  - Argument variable
  - for handling command-line parameters

Command-line parameters

```
from sys import argv

script, first, second, third = argv

print 'The script is called:', script
print 'Your first variable is:', first
print 'Your second variable is:', second
print 'Your third variable is:', third
```



```
$ python cli_parameters.py one two 3
The script is called:  cli_parameters.py
Your first variable is:  one
Your second variable is:  two
Your third variable is:  3
```



```
$ python cli_parameters.py
Traceback (most recent call last):
  File "cli_parameters.py", line 23, in <module>
    script, first, second, third = argv #argv unpacking
ValueError: need more than 1 value to unpack
```

Functions

- A **function** is a named sequence of statements that performs a computation
  - Definition first:
    - specify the name and the sequence of statements
  - Then usage:
    - “call” the function by name
- Examples
  - Type conversion functions
    - int(‘32’) → 32
    - str(3.2479) → ‘3.2479’

Math functions

- Located in the math module

```
import math

signal_power = 10.0
noise_power = 0.01
ratio = signal_power / noise_power
print "ratio:", ratio

decibels = 10 * math.log10(ratio)
print "decibels:", decibels

radians = 0.7
height = math.sin(radians)
print height
```

Function call

String functions

- len()
  - Gets the length (the number of characters) of a string
- lower()
  - Gets rid of all the capitalization in a string
- upper()
  - Transform a string in upper case
- str()
  - Transform «everything» in a string

String functions: an example

```
course_name = 'Ambient Intelligence'

string_len = len(course_name)
print string_len # 20

print course_name.lower() # ambient intelligence

print course_name.upper() # AMBIENT INTELLIGENCE

pi = 3.14
print "the value of pi is around " + str(pi)
```

without str()
it gives an error

New functions

- Can be defined by developers
- Typically used to group homogeneous code portions
  - i.e., code for accomplishing a well-defined operation
- Enable re-use
  - Same operation can be re-used several times
- Defined using the keyword **def**

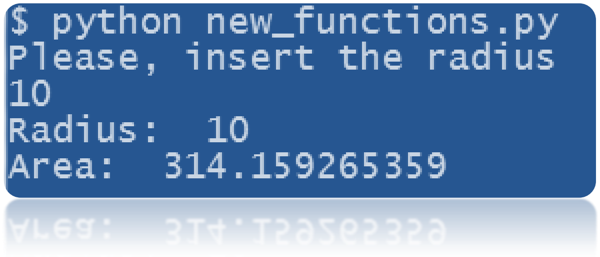
New functions

- Compute the area of a disk, given the radius

```
import math

def circle_area(radius):
   return radius**2*math.pi

radius = raw_input('Please, insert the radius\n')
print 'Radius: ', radius
print 'Area: ', circle_area(radius)
```

Function definition

Function call



```
$ python new_functions.py
Please, insert the radius
10
Radius:  10
Area:  314.159265359
```

Docstring

- Optional, multiline comment
- Explains what the function does
- Starts and ends with """ or '''

```
import math

def circle_area(radius):
    '''Compute the circle area given its radius'''
    return radius**2*math.pi

radius = raw_input('Please, insert the radius\n')
print 'Radius: ', radius
print 'Area: ', circle_area(radius)
```

Modules

- A way to logically organize the code
- They are files consisting of Python code
  - they can define (and implement) functions, variables, etc.
  - typically, the file containing a module is called in the same way
    - e.g., the module *math* resides in a file named *math.py*
- We already met them

```
import math

from sys import argv
```

Importing modules

- **import** *module_name*
  - allows to use all the items present in a module

```
import math

def circle_area(radius):
   return radius**2*math.pi

…
```

Import the *math* module

Call the *pi* variable from the *math* module

Importing modules

- **from** *module_name* **import** *name*
  - it only imports *name* from the specified module

```
from math import pi

def circle_area(radius):
   return radius**2*pi
...
```

Import *pi* from the *math* module

Use the *pi* variable

- **from** *module_name* **import** *
  - it imports all names from a module
  - **do not use!**

Playing with files

- Python script can read and write files
- First, open a file
  - You can use the open() function
- Then, you can read or write it
  - With read(), readline(), or write()
- Finally, remember to close the file
  - You can use the close() function

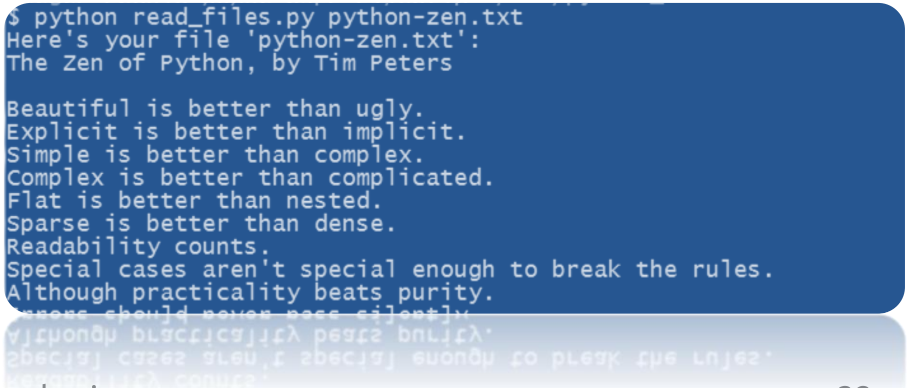
Reading files

- Read a file taking its name from command line

```
from sys import argv

filename = argv[1]
txt = open(filename)

print “Here’s your file %r:", % filename
print txt.read()

print “\nType the filename again:”
file_again = raw_input(“> ”)
txt_again = open(file_again)
print txt_again.read()
```

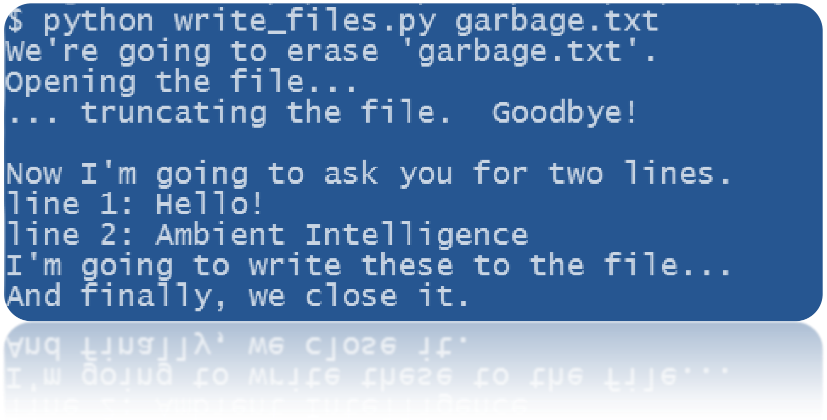
Open the file

Show the file content



```
$ python read_files.py python-zen.txt
Here's your file 'python-zen.txt':
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
```

Writing files



```
$ python write_files.py garbage.txt
We're going to erase 'garbage.txt'.
Opening the file...
... truncating the file.  Goodbye!

Now I'm going to ask you for two lines.
line 1: Hello!
line 2: Ambient Intelligence
I'm going to write these to the file...
And finally, we close it.
```

```
from sys import argv

script, filename = argv

print "We're going to erase %r." % filename
print "Opening the file..."
target = open(filename, 'w')
print "... truncating the file. Goodbye!"
target.truncate()
print "\nNow I'm going to ask you for two lines."
line1 = raw_input("line 1: ")
line2 = raw_input("line 2: ")
print "I'm going to write these to the file."
target.write(line1)
target.write("\n")
target.write(line2)
target.write("\n")
print "And finally, we close it."
target.close()
```

Open the file in **write** mode

Empties the file

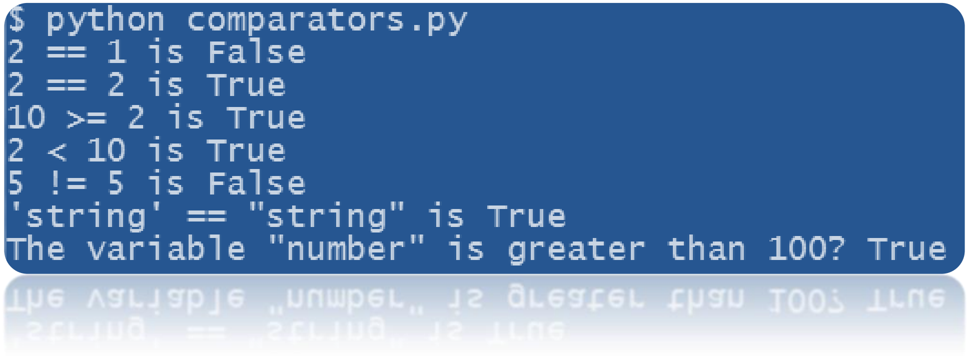
Write a string to the file

Conditionals and control flow

- Control flow gives the ability to choose among outcomes
  - based off what else is happening in the program
- Comparators
  - Equal to → **==**
  - Not equal to → **!=**
  - Less than → **<**
  - Less than or equal to → **<=**
  - Greater than → **>**
  - Greater than or equal to → **>=**

Comparators: an example

```
print 2 == 1 # False

print 2 == 2 # True

print 10 >= 2 # True

print 2 < 10 # True

print 5 != 5 # False

print 'string' == "string" # True

number = 123
print number > 100 # True
```



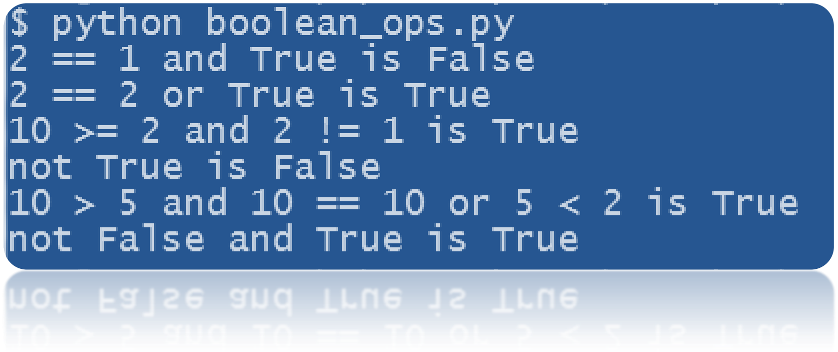
```
$ python comparators.py
2 == 1 is False
2 == 2 is True
10 >= 2 is True
2 < 10 is True
5 != 5 is False
'string' == "string" is True
The variable "number" is greater than 100? True
```

Boolean operators

- They are three:
  - **not**
  - **and**
  - **or**
- Not evaluated from left to right
  - not is evaluated **first**
  - and is evaluated **next**
  - or is evaluated **last**

Boolean operators: an example

```
print 2 == 1 and True # False

print 2 == 2 or True # True

print 10 >= 2 and 2 != 1 # True

print not True # False

print 10 > 5 and 10 == 10 or 5 < 2 # True

print not False and True # True
```



```
$ python boolean_ops.py
2 == 1 and True is False
2 == 2 or True is True
10 >= 2 and 2 != 1 is True
not True is False
10 > 5 and 10 == 10 or 5 < 2 is True
not False and True is True
```

Conditional statement

- **if** is a statements that executes some code after checking if a given expression is *True*

- Structure

  if *expression*:

  do something

```
people = 20
cats = 30

if people < cats:
    print 'Too many cats! The world is doomed!'

if people > cats:
    print 'Not many cats! The world is saved!'
```

More “if”

- Let’s try to “improve” the previous example

```
people = 20
cats = 30
if people < cats:
   print 'Too many cats! The world is doomed!'
elif people > cats:
   print 'Not many cats! The world is saved!'
else:
   print "We can’t decide.”
```

else if

- Chained conditionals
  - To express more than two possibilities
  - Each condition is checked in order

Loops and lists

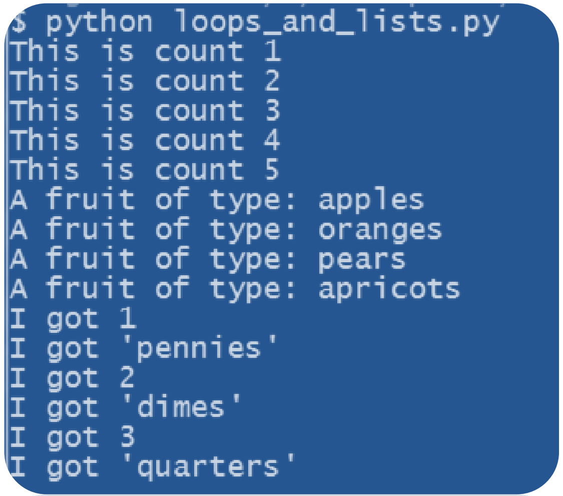
- Loop
  - An easy way to do repetitive things
  - A condition to start and stop the loop is required
  - e.g., **for** and **while** loops
- List
  - A datatype for storing multiple items
    - a sequence of values
  - You can assign items to a list in this way:

    list_name = [item1, item2, …]

Loops and lists: an example

```
the_count = [1, 2, 3, 4, 5]
fruits = ['apples', 'oranges', 'pears', 'apricots']
change = [1, 'pennies', 2, 'dimes', 3, 'quarters']

# this first kind of for-loop goes through a list
for number in the_count:
    print 'This is count %d' % number

# same as above
for fruit in fruits:
    print 'A fruit of type: %s' % fruit

# we can go through mixed lists too
# notice that we have to use %r since we don't know what's in it
for i in change:
    print 'I got %r' % i
```

Three lists



```
$ python loops_and_lists.py
This is count 1
This is count 2
This is count 3
This is count 4
This is count 5
A fruit of type: apples
A fruit of type: oranges
A fruit of type: pears
A fruit of type: apricots
I got 1
I got 'pennies'
I got 2
I got 'dimes'
I got 3
I got 'quarters'
```

Loops and lists: an example

```
the_count = [1, 2, 3, 4, 5]
fruits = ['apples', 'oranges', 'pears', 'apricots']
change = [1, 'pennies', 2, 'dimes', 3, 'quarters']

# this first kind of for-loop goes through a list
for number in the_count:
    print 'This is count %d' % number

# same as above
for fruit in fruits:
    print 'A fruit of type: %s' %

# we can go through mixed lists too
# notice that we have to use %r since we don't know what's in it
for i in change:
    print 'I got %r' % i
```

Structure of a for loop

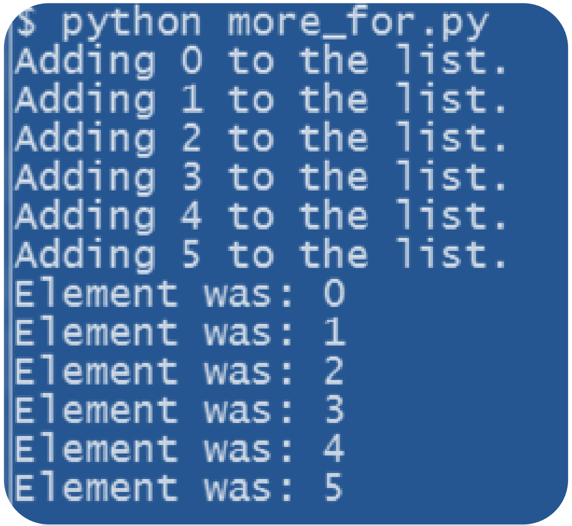
- *for* variable *in* collection*:*
- *indent* for the loop body

More “for”

```
# we can also build lists: start with an empty one…
elements = []

# then use the range function to do 0 to 5 counts
for i in range(0, 6):
    print 'Adding %d to the list.' % i
    # append() is a function that lists understand
    elements.append(i)

# now we can print them out
for i in elements:
    print 'Element was: %d' % i
```

Empty list

Repeat 6 times

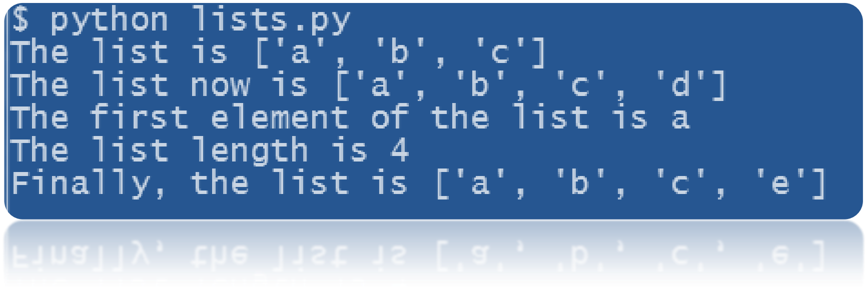


```
$ python more_for.py
Adding 0 to the list.
Adding 1 to the list.
Adding 2 to the list.
Adding 3 to the list.
Adding 4 to the list.
Adding 5 to the list.
Element was: 0
Element was: 1
Element was: 2
Element was: 3
Element was: 4
Element was: 5
```

Lists

- Mutable
- Do not have a fixed length
  - You can add items to a list at any time
- Accessible by index

```
letters = [‘a’, ‘b’, ‘c’]
letters.append(‘d’)
print letters # a, b, c, d

print letters[0] # a

print len(letters) # 4

letters[3] = ‘e’
print letters # a, b, c, e
```

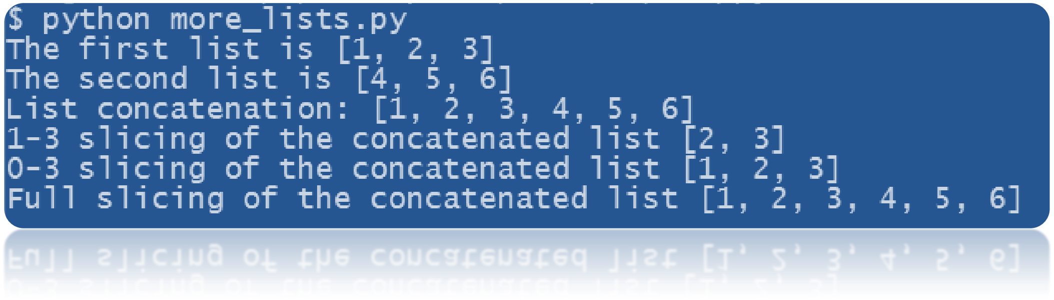


```
$ python lists.py
The list is ['a', 'b', 'c']
The list now is ['a', 'b', 'c', 'd']
The first element of the list is a
The list length is 4
Finally, the list is ['a', 'b', 'c', 'e']
```

More lists

- List concatenation
  - with the + operator

```
a = [1, 2, 3]
b = [4, 5, 6]
c = a + b
print c # 1, 2, 3, 4, 5, 6
```



```
$ python more_lists.py
The first list is [1, 2, 3]
The second list is [4, 5, 6]
List concatenation: [1, 2, 3, 4, 5, 6]
1-3 slicing of the concatenated list [2, 3]
0-3 slicing of the concatenated list [1, 2, 3]
Full slicing of the concatenated list [1, 2, 3, 4, 5, 6]
```

- List slices
  - to access a portion of a list
  - with the [:] operator

```
c = [1, 2, 3, 4, 5, 6]
d = c[1:3] # d is [2, 3]
e = c[:3] # e is [1, 2, 3]
f = c[:] # f is a full copy of c
```

More lists

- List concatenation
  - with the + operator

```
a = [1, 2, 3]
b = [4, 5, 6]
c = a + b
print c # 1, 2, 3, 4, 5, 6
```

- List slices
  - to access a portion of a list
  - with the [:] operator

```
c = [1, 2, 3, 4, 5, 6]
d = c[1:3] # d is [2, 3]
e = c[:3] # e is [1, 2, 3]
f = c[:] # f is a full copy of c
```

works with string, too

List functions

- append()
  - add a new element to the end of a list
  - e.g., *my_list.append('d')*
- sort()
  - arrange the elements of the list from low to high
  - e.g., from a to z, from 1 to infinite, etc.
- extend()
  - takes a list as an argument and appends all its elements
  - e.g., *first_list.extend(second_list)*

Deleting elements from a list

- Several ways to delete elements from a list
- If you know the index of the element to remove: **pop()**
  - without providing an index, pop() delete the last element
- If you know the element to remove (but not the index): **remove()**
- To remove more than one element: **del()**
  - with a slice index
    - e.g., *del my_list[5:8]*

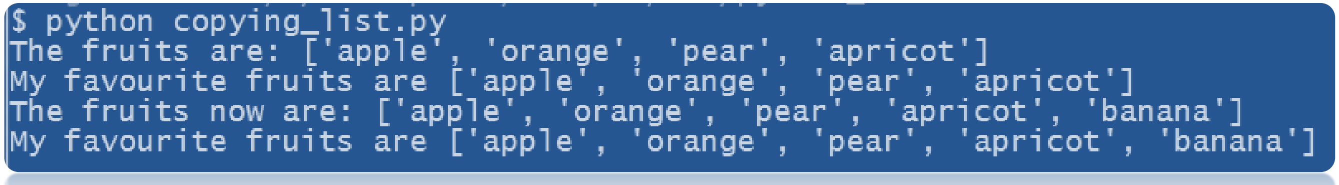
Strings vs. lists

- A string is a sequence of character, but a list of character is not a string
- To convert a string into a list of characters: **list()**
  - e.g., *my_list = list(my_string)*
- To break a string into separate words: **split()**
  - split a list according to some delimiters (default: *space*)
  - e.g., *my_list = my_string.split()*
  - The inverse function is **join()**

Copying lists

- What happens here?

```
fruits = ['apple', 'orange', 'pear', 'apricot']
print 'The fruits are:', fruits
favourite_fruits = fruits
print 'My favourite fruits are', favourite_fruits

# add a fruit to the original list
fruits.append(‘banana’)

print 'The fruits now are:', fruits
print 'My favourite fruits are', favourite_fruits
```



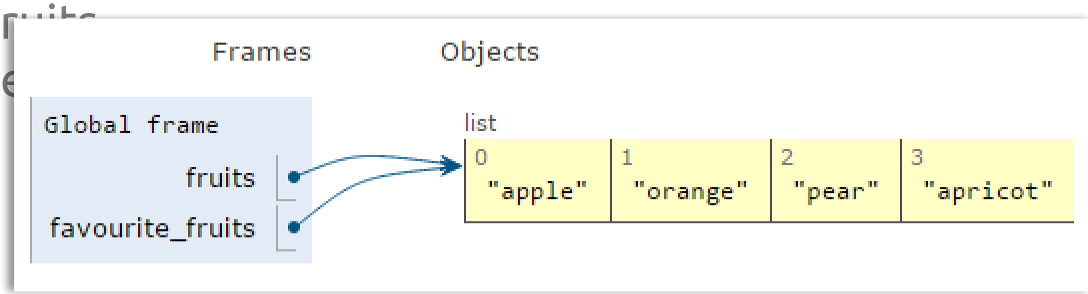
```
$ python copying_list.py
The fruits are: ['apple', 'orange', 'pear', 'apricot']
My favourite fruits are ['apple', 'orange', 'pear', 'apricot']
The fruits now are: ['apple', 'orange', 'pear', 'apricot', 'banana']
My favourite fruits are ['apple', 'orange', 'pear', 'apricot', 'banana']
```

Copying lists

- What happens here?

```
fruits = ['apple', 'orange', 'pear', 'apricot']
print 'The fruits are:', fruits
favourite_fruits = fruits
print 'My favourite fruits are', favourite_fruits

# add a fruit to the original list
fruits.append(‘banana’)

print 'The fruits now are:', fr
print 'My favourite fruits are
```

We **do not** make a copy of the entire list, but we only make a **reference** to it!



Copying lists

- How to make a **full** copy of a list?
- Various methods exist
  - you can entirely slice a list
    - *favourite_fruits = fruits[:]*
  - you can create a new list from the existing one
    - *favourite_fruits = list(fruit)*
  - you can extend an empty list with the existing one
    - *favourite_fruits.extend(fruit)*
- Prefer the **list()** method, when possible!

Dictionaries

- Similar to lists, but you can access values by looking up a key instead of an index
  - A key can be a string or a number
- Example
  - A dictionary with 3 key-value pairs

    dict = { ‘key1’ : 1, ‘key2’ : 2, ‘key3’ : 3 }
- Mutable, like lists
  - They can be changed after their creation

Dictionaries: an example

```
# create a mapping of U.S. state to abbreviation
states = {
    'Oregon' : 'OR',
    'Florida' : 'FL',
    'California' : 'CA'
}
print 'States:', states
print 'Is Oregon available?', 'Oregon' in states

# add some more states
states['New York'] = 'NY'
states['Michigan'] = 'MI'

# print two states
print "New York’s abbreviation is: ", states[‘New York’]
print "Florida’s abbreviation is: ", states[‘Florida’]
```

Create a dictionary with 3 key-value pairs

Add two more key-value pairs

More dictionaries

```
# states is a dictionary defined as before

# print every state abbreviation
for state, abbrev in states.items():
    print "%s is abbreviated %s", % (state, abbrev)

# safely get an abbreviation of a state that might not be there
state = states.get('Texas', None) # None is the default

if not state:
    print "Sorry, no Texas."

# get a state abbreviation with a default value
next_state = states.get('Massachusetts', 'Does Not Exist')
print "Massachusetts is abbreviated %s", % next_state
```

Dictionary functions

- len()
  - dictionary length: the number of key-value pairs
- del()
  - remove a key-value pair
    - e.g., *del my_dict[my_key]*
- clear()
  - remove all items from a dictionary
- keys() and values()
  - return a copy of the dictionary's list of key and value, respectively

References and Links

- The Python Tutorial, http://docs.python.org/2/tutorial/
- «*Think Python: How to think like a computer scientist*», Allen Downey, Green Tea Press, Needham, Massachusetts
- «*Dive into Python 2*», Mark Pilgrim
- «Learn Python the Hard Way», Zed Shaw
- «Learning Python» (5th edition), Mark Lutz, O'Reilly
- The Google Python course, https://developer.google.com/edu/python
- Online Python Tutor, http://pythontutor.com

Questions?

01PRD Ambient Intelligence: Technology and Design

Luigi De Russis and Dario Bonino
luigi.derussis@polito.it
bonino@isbm.it

License