Introduction to Python

AARHUS UNIVERSITET
INSTITUT FOR INGENIØRVIDENSKAB

MIRGITA FRASHERI | POSTDOC

What is Python?

Python is an interpreted programming language that allows you to do almost anything possible with a compiled language (C/C++/Fortran) without requiring all the complexity.

Slides adapted from Nicolás Navarro-Guerrero

Main Features

- Automatic garbage collection

Main Features

- Automatic garbage collection
- Interpreted and interactive

Main Features

- Automatic garbage collection
- Interpreted and interactive
- Object-oriented

Main Features

- Automatic garbage collection
- Interpreted and interactive
- Object-oriented
- Useful built-in types

Main Features

- Automatic garbage collection
- Interpreted and interactive
- Object-oriented
- Useful built-in types
- Easy matrix algebra (via numpy)

Main Features

- Automatic garbage collection
- Interpreted and interactive
- Object-oriented
- Useful built-in types
- Easy matrix algebra (via numpy)
- Easy to program GUIs

Main Features

- Automatic garbage collection
- Interpreted and interactive
- Object-oriented
- Useful built-in types
- Easy matrix algebra (via numpy)
- Easy to program GUIs
- Lot of documentation, tutorials and libraries

A Sample of Code

```
x = 4 - 1.0      # comment: integer difference
y = "Hello"      # double quotes
y = 'Hello'      # single quotes also work
```

A Sample of Code

```
x = 4 - 1.0     # comment: integer difference
y = "Hello"     # double quotes
y = 'Hello'     # single quotes also work


if x == 0 or y == "Hello":
    x= x+ 1
    y = y + " World" # concatenating two strings
print(y)
print(y * 3)   # repeating a string
len(y)         # String length
```

Language Introduction

- Assignment uses = and comparison uses ==

Language Introduction

- Assignment uses = and comparison uses ==
- + – * / % compute numbers as expected

Language Introduction

- Assignment uses = and comparison uses ==
- + − * / % compute numbers as expected
- Use + for string concatenation

Language Introduction

- Assignment uses = and comparison uses ==
- + – * / % compute numbers as expected
- Use + for string concatenation
- Logical operators are words (and, or, not), but not symbols (&&, ||, !)

Language Introduction

- Assignment uses = and comparison uses ==
- + – * / % compute numbers as expected
- Use + for string concatenation
- Logical operators are words (and, or, not), but not symbols (&&, ||, !)
- First assignment to a variable will create it

Language Introduction

- Assignment uses = and comparison uses ==
- + – * / % compute numbers as expected
- Use + for string concatenation
- Logical operators are words (and, or, not), but not symbols (&&, ||, !)
- First assignment to a variable will create it
- Python assigns the variable types

Tips on Code Style

- Use consistent indentation to mark blocks of code

Tips on Code Style

- Use consistent indentation to mark blocks of code
- Use a newline to end a line of code or use \ when must go to next line prematurely)

Tips on Code Style

- Use consistent indentation to mark blocks of code
- Use a newline to end a line of code or use \ when must go to next line prematurely)
- Comments start with # – the rest of line is ignored

Tips on Code Style

- Use consistent indentation to mark blocks of code
- Use a newline to end a line of code or use \ when must go to next line prematurely)
- Comments start with # – the rest of line is ignored
- A documentation string can be included as the first line of any function or class with triple double-quotes

Basic Data Types

- Integers (default for numbers)

Basic Data Types

- Integers (default for numbers)
- Strings
  - Can use ” ” or ‘’
  - Unmatched quotes can occur in the string: "matt’s"
  - Use triple double-quotes for multi-line strings or strings which contain both ’ and " inside: ” ” ” a’ b” c ” ” ”

Basic Data Types

- Integers (default for numbers)

- Strings
  - Can use ” ” or ‘’
  - Unmatched quotes can occur in the string: "matt’s"
  - Use triple double-quotes for multi-line strings or strings which contain both ’ and " inside: ” ” ” a’ b” c ” ” ”

- Dynamic Typing (Python determines data types automatically),
  - But Python is not casual about types, it enforces them thereafter: Strong Typing
  - e.g., you can’t just append an integer to a string.

Naming Rules

- Names are case sensitive and cannot start with a number. They can contain letters, numbers, and underscores
  turtlebot Turtlebot _turtlebot _2_turtlebot turtlebot_2 TURTLEBOT

Naming Rules

- Names are case sensitive and cannot start with a number. They can contain letters, numbers, and underscores
  turtlebot Turtlebot _turtlebot _2_turtlebot turtlebot_2 TURTLEBOT

- There are some reserved words:
  and, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, while

List Objects

- List creation with brackets
```
lst = [10,11,12,13,14]
```

List Objects

- List creation with brackets

```
lst = [10,11,12,13,14]
```

- Concatenating list

```
[10, 11] + [12,13] # simply use the + operator
```

List Objects

- List creation with brackets

```
lst = [10,11,12,13,14]
```

- Concatenating list

```
[10, 11] + [12,13] # simply use the + operator
```

- Repeating elements in lists

```
[10, 11] * 2 # produces [10, 11, 10, 11]
```

List Objects

- List creation with brackets
  `lst = [10,11,12,13,14]`

- Concatenating list
  `[10, 11] + [12,13] # simply use the + operator`

- Repeating elements in lists
  `[10, 11] * 2 # produces [10, 11, 10, 11]`

- range(start, stop, step)

```
range(5)          # [0, 1, 2, 3,   4]
range(2,7)        # [2, 3, 4, 5,   6]
range(2,7,2)      # [2, 4, 6]
```

Indexing

- Retrieving and element

```
lst = [10,11,12,13,14]
lst[0]          # produces 10
```

Indexing

- Retrieving and element

```
lst = [10,11,12,13,14]
lst[0]          # produces 10
```

- Setting an element

```
lst[1] = 21    # produces [10,21,12,13,14]
```

Indexing

- Retrieving and element

```
lst = [10,11,12,13,14]
lst[0]          # produces 10
```

- Setting an element

```
lst[1] = 21     # produces [10,21,12,13,14]
```

- Out of bounds

```
lst[10]         # raises and error
```

Indexing

- Retrieving and element

```
lst = [10,11,12,13,14]
lst[0]          # produces 10
```

- Setting an element

```
lst[1] = 21     # produces [10,21,12,13,14]
```

- Out of bounds

```
lst[10]         # raises and error
```

- negative indices count backward from the end of the list

```
lst[-1]         # produces 14
```

Assignment

- Multiple Assignment

```
x, y, z = 1, 2, 3 # y = 2
```

Assignment

- Multiple Assignment

```
x, y, z = 1, 2, 3 # y = 2
```

- Assignment creates object references

```
a = [0,1,2]

b = a          #  x and y point at the same list
b[1] = 6       #  changes to y also change x
print(a)
print(b)
b = [3, 4]     #  re-assigning b to a new list
               #  decouples the two lists
```

If Statements

- if/elif/else provide conditional execution of code blocks

```
x = 10
if x > 0:
  print(1)
elif x == 0:
  print(0)
else:
  print(-1)
```

If Statements

- if/elif/else provide conditional execution of code blocks

```
x = 10
if x > 0:
  print(1)
elif x == 0:
  print(0)
else:
  print(-1)
```

- elif and else are not mandatory

If Statements

- if/elif/else provide conditional execution of code blocks

```
x = 10
if x > 0:
  print(1)
elif x == 0:
  print(0)
else:
  print(-1)
```

- elif and else are not mandatory
- True means any non-zero number or non-empty object

If Statements

- if/elif/else provide conditional execution of code blocks

```
x = 10
if x > 0:
  print(1)
elif x == 0:
  print(0)
else:
  print(-1)
```

- elif and else are not mandatory
- True means any non-zero number or non-empty object
- False means not true: zero, empty object, or None

For Loops

- For loops iterate over a sequence of objects.

```
for i in range(5):
    print(i) # produces 0 1 2 3 4
```

For Loops

- For loops iterate over a sequence of objects.

```
for i in range(5):
    print(i) # produces 0 1 2 3 4

for i in 'abcde':
    print(i)
# produces  a b c d e
```

For Loops

- For loops iterate over a sequence of objects.

```
for i in range(5):
    print(i) # produces 0 1 2 3 4

for i in 'abcde':
    print(i)
# produces a b c d e

lst=['dogs','cats','bears']
for item in lst:
    print item + ' '
# produces dogs cats bears
```

While Loops

- While loops iterate until a condition is met.

```
lst = range(3) while lst:
    print(lst)
    lst = lst[1:]

# produces
# [0, 1, 2]
# [1, 2]
# [2]
```

While Loops

- While loops iterate until a condition is met.

```
lst = range(3) while lst:
    print(lst)
    lst = lst[1:]

# produces
# [0, 1, 2]
# [1, 2]
# [2]
```

- `break` can be used to breaking out of a loop

Functions

```
def add ( arg 0 ,   arg ):
  a  =  arg 0  +  arg 1
  return a
```

Functions

```
def add ( arg 0 ,   arg ):
    a = arg 0 + arg 1
    return a
```

- The keyword def indicates the start of a function

Functions

```
def add ( arg 0 ,   arg ):
  a  =  arg 0  +  arg 1
  return a
```

- The keyword def indicates the start of a function
- Function arguments are listed separated by commas (by assignment)

Functions

```
def add ( arg 0 ,   arg ):
   a  =  arg 0  +  arg 1
  return a
```

- The keyword def indicates the start of a function
- Function arguments are listed separated by commas (by assignment)
- A colon (:) terminates the function definition

Functions

```
def add ( arg 0 ,   arg ):
    a  =  arg 0  +  arg 1
   return a
```

- The keyword def indicates the start of a function
- Function arguments are listed separated by commas (by assignment)
- A colon (:) terminates the function definition
- Indentation is used to indicate the contents of the function (not optional)

Functions

```
def add ( arg 0 ,   arg ):
   a  =  arg 0  +  arg 1
  return a
```

- The keyword def indicates the start of a function
- Function arguments are listed separated by commas (by assignment)
- A colon (:) terminates the function definition
- Indentation is used to indicate the contents of the function (not optional)
- return is optional. If omitted, it takes the special value None

Classes

```
class stack():
    def __init__(self):
        self.items = []

    def push(self, x):
        self.items.append(x)

    def pop(self):
        x= self.items[-1] del sel
        f.items[-1] return x

    def empty(self):
        return len(self.items) == 0
```

Classes

```
class stack():
  def __init__(self):
    self.items = []

  def push(self, x):
    self.items.append(x)

  def pop(self):
    x= self.items[-1] del sel
    f.items[-1] return x

  def empty(self):
    return len(self.items) == 0
```

Usage:

```
t = stack()
print t.empty()
t.push("hello")
print t.empty()
t.pop()
print t.empty()
```

Modules

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# today.py
import datetime
today = datetime.date.today()
print today
```

Modules

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# today.py
import datetime
today = datetime.date.today()
print today
```

Run from a terminal

```
python today.py

today.py
```

Setting Up PYTHONPATH

- PYTHONPATH is an environment variable (or set of registry entries on Windows) that lists the directories Python searches for modules (UNIX – .bashrc)

- export PYTHONPATH=${PYTHONPATH}:/path_to_library