

Introduction to Python

- Installation and Crash Course -

2017 – 2019 - 2020

Ando Ki
Future Design Systems
adki@future-ds.com / www.future-ds.com

Table of contents

- ▣ What is Python
- ▣ Python installation
 - ◆ Native
 - ◆ Virtualenv
 - ◆ Anacoda
- ▣ Running Python
- ▣ Python language elements
 - ◆ comment, identifiers, Keywords
 - ◆ Data types, string, list, tuple, dictionary, set
 - ◆ Operators, control flow
 - ◆ Function, argument passing
 - ◆ Module, input and output
- ▣ Python C-API
- ▣ Python extension methods
- ▣ Manual wrapping C/C++ for Python

What is Python?



- Python is a programming language freely available from www.python.org.
- Python is an interpreted, interactive, object-oriented programming language.
 - interpreted: it is processed at runtime by the interpreter
 - interactive: interact with the interpreter directly
 - object-oriented: encapsulates code within objects

Python version

- Python 2.7: ~2010
- Python 3.x: 2008~
- Python 3.7.4: 2019
 - not backward compatible:
 - <https://docs.python.org/3/whatsnew/3.0.html>



Apollo

Python (dragon or snake)

Delphi (town in Greece)



Copyright © 2016-2019 by Ando Ki

(3)

FUTURE
Design Systems

Python package installation

Ways of installing python

- PIP: pip is a tool for installing Python packages from the Python Package Index.
 - Python Package Index (PyPI) – The official third-party software repository for the Python programming language; <https://pypi.python.org/pypi>

Virtualenv

- Python has version dependencies and directory sensitivity
 - To deal with this issue, use virtual environment
 - Virtualenv is a tool to isolate python environment
 - or use 'import __future__'
 - a pseudo-module which programmers can use to enable new language features which are not compatible with the current interpreter.

- pip:** the Python Package Manager. (pip is not included with python by default.)
- pyenv:** Python Version Manager (easily switch between multiple versions of Python)
- virtualenv:** Python Environment Manager.
- Anaconda:** Package Manager + Environment Manager + Additional Scientific Libraries.

The main purpose of Python virtual environments is to create an isolated environment for Python projects, since each project can have its own dependencies.

Copyright © 2016-2019 by Ando Ki

(4)

FUTURE
Design Systems

Python installation: reference

❏ Unix and Linux

- ◆ <https://www.python.org/downloads/>
- ◆ interpreter: `/usr/local/bin/python`
- ◆ library: `/usr/local/lib/pythonXX`
 - ❖ XX: version
 - ▢ Python 2.7: ~2010
 - ▢ Python 3.x: 2008~

❏ Windows

- ◆ <https://www.python.org/downloads/>

❏ Environment variables

- ◆ **PYTHONPATH**
 - ❖ where python interpreter, module files, and source code
- ◆ **PYTHONSTARTUP**
 - ❖ where initialization file
- ◆ **PYTHONCASEOK**
 - ❖ for Windows, find out case-insensitive match
- ◆ **PYTHONHOME**
 - ❖ Alternative module search path
 - ❖ It may be embedded in 'PYTHONPATH' or 'PYTHONSTARTUP'

Table of contents

❏ What is Python

❏ Python installation

- ◆ Native
- ◆ Virtualenv
- ◆ Anaconda

❏ Running Python

❏ Python language elements

- ◆ comment, identifiers, Keywords
- ◆ Data types, string, list, tuple, dictionary, set
- ◆ Operators, control flow
- ◆ Function, argument passing
- ◆ Module, input and output

❏ Python C-API

❏ Python extension methods

❏ Manual wrapping C/C++ for Python

Python installation

On Ubuntu 16.04

- ◆ \$ sudo add-apt-repository ppa:deadsnakes/ppa
- ◆ \$ sudo apt update
- ◆ \$ sudo apt install python3.6

On Ubuntu 16.10 and 17.04 and 18.04

- ◆ \$ sudo apt update
- ◆ \$ sudo apt install python3.6

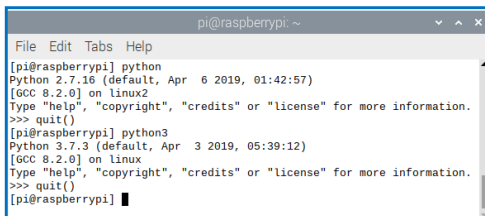
Python installation

On Ubuntu

- ◆ for Python 3
 - ◆ \$ sudo apt update
 - ◆ \$ sudo apt install python3.6
 - ◆ \$ sudo apt install python3-pip
 - ◆ \$ pip3 --version
 - ◆ \$ pip3 list
 - ◆ \$ pip3 install package_name[==version]
 - ◆ \$ pip3 install -upgrade package_name
 - ◆ \$ pip3 uninstall package_name
- ◆ for Python 2
 - ◆ \$ sudo apt update
 - ◆ \$ sudo apt install python-pip
 - ◆ \$ pip --version
 - ◆ \$ pip list
 - ◆ \$ pip install -upgrade package_name
 - ◆ \$ pip uninstall package_name

On Raspberry Pi

- ◆ Python 2.6 and python 3.5 are installed by default on Raspbian (not lite version).
- ◆ Check Python using 'python' and 'python3' commands.



```
pi@raspberrypi: ~  
File Edit Tabs Help  
[pi@raspberrypi] python  
Python 2.7.16 (default, Apr 6 2019, 01:42:57)  
[GCC 8.2.0] on linux2  
Type "help", "copyright", "credits" or "license" for more information.  
>>> quit()  
[pi@raspberrypi] python3  
Python 3.7.3 (default, Apr 3 2019, 05:39:12)  
[GCC 8.2.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> quit()  
[pi@raspberrypi]
```

- ◆ To install Python3, if not installed yet.

\$ sudo apt-get install python3

Table of contents

- ▣ What is Python
- ▣ Python installation
 - ◆ Native
 - ◆ Virtualenv
 - ◆ Anacoda
- ▣ Running Python
- ▣ Python language elements
 - ◆ comment, identifiers, Keywords
 - ◆ Data types, string, list, tuple, dictionary, set
 - ◆ Operators, control flow
 - ◆ Function, argument passing
 - ◆ Module, input and output
- ▣ Python C-API
- ▣ Python extension methods
- ▣ Manual wrapping C/C++ for Python

Python installation with Virtualenv on Ubuntu

- ▣ Step 1: Install pip and virtualenv (for Python 2.7)
- ▣ Step 2: Create a virtual environment
- ▣ Step 3: Activate virtualenv

Python installation with Virtualenv on Ubuntu

```
$ sudo apt-get update
$ sudo apt-get install python-pip python-dev python-virtualenv
$ virtualenv --system-site-packages ~/my_python
$ source ~/my_python/bin/activate
(my_python)$
(my_python)$ python --version
Python 2.7.6
....
....
(my_python)$ deactivate
```

Install pip and virtualenv

Create a virtual env

Activate the virtualenv

Virtualenv prompt

Check version of Python

Note your current working directory will be the directory where 'activate' called.

To uninstall Python, simply remove the directory as follows,
\$ /bin/rm -rf ~/my_python

Running Python code using Virtualenv

```
$ source ~/my_python/bin/activate
(my_python)$ vi hello.py
print "Hello World!"
(my_python)$ python hello.py
Hello World!
(my_python)$ deactivate
$
```

Activate the virtualenv
Note that '~/my_python' is
where the Python virtual
environment has been built.

Prepare Python code

Running a Python script

Exit from virtualenv

Get back to normal shell

```
# Python 'hello.py'
print "Hello World!"
```

Python testing after installation

Table of contents

- ❑ What is Python
- ❑ Python installation
 - ◆ Native
 - ◆ Virtualenv
 - ◆ Anaconda
- ❑ Running Python
- ❑ Python language elements
 - ◆ comment, identifiers, Keywords
 - ◆ Data types, string, list, tuple, dictionary, set
 - ◆ Operators, control flow
 - ◆ Function, argument passing
 - ◆ Module, input and output
- ❑ Python C-API
- ❑ Python extension methods
- ❑ Manual wrapping C/C++ for Python

Install Anaconda2

- ❑ Download Anaconda2
 - ◆ `$ cd; $ mkdir tmp; $ cd tmp`
 - ◆ `$ wget https://repo.anaconda.com/archive/Anaconda2-2.5.0-Linux-x86_64.sh`
- ❑ Run the installer (Installer requires bzip, please install it if you don't have it)
 - ◆ `$ bash ./Anaconda2-5.1.0-Linux-x86_64.sh -p ${HOME}/anaconda2`
- ❑ Ensure that your `.bashrc` is preparing Anaconda, by including these lines
 - ◆ `~/.bashrc: export PATH=${HOME}/anaconda2/bin:$PATH`
 - # added by Anaconda2 installer (at `.bashrc` file)
 - `export PATH="/home/username/anaconda2/bin:$PATH"`
 - `./home/username/anaconda2/etc/profile.d/conda.sh`
- ❑ To remove Anaconda2
 - ◆ `$ conda install anaconda-clean`
 - ◆ `$ anaconda-clean --yes`
- ❑ Then remove directories
 - ◆ `$ /bin/rm -rf ~/anaconda2`
 - ◆ `$ /bin/rm -rf ~/.anaconda_backup`

Do not use "`${HOME}`" for `"/home/username"`
where `'/home/username'` should be your home
directory.

Check Anaconda2

- ❑ After updating the bashrc, source it to load the new anaconda path or open a new terminal (\$ source ~/.bashrc)

- ◆ ~/.bashrc: . /\${HOME}/anaconda2/etc/profile.d/conda.sh

- ❑ Check conda

- ◆ \$ conda -V

- ◆ 3.19.1

Prepare your virtual environment

- ❑ Make a virtual environment for my_env

- ◆ \$ conda create --name *my_env* python=2.7

Choose name for your environment

- ❑ Ensure that your .bashrc is preparing Anaconda, by including these lines

- ◆ ~/.bashrc: . /\${HOME}/anaconda2/etc/profile.d/conda.sh

- ❑ Activate conda and add packages

- ◆ \$ conda activate *my_env*

- ◆ (*my_env*) \$ python

- ◆ // python commands

- ◆ >>> quit()

- ◆ (*my_env*) \$ conda deactivate

// To check packages in the conda
\$ conda list

// To check conda environment
\$ conda env list

// To remove conda environment
\$ conda-env remove -n my_env
or
\$ conda remove --name my_env --all

Table of contents

- ❏ What is Python
- ❏ Python installation
 - ◆ Native
 - ◆ Virtualenv
 - ◆ Anaconda
- ❏ Running Python
- ❏ Python language elements
 - ◆ comment, identifiers, Keywords
 - ◆ Data types, string, list, tuple, dictionary, set
 - ◆ Operators, control flow
 - ◆ Function, argument passing
 - ◆ Module, input and output
- ❏ Python C-API
- ❏ Python extension methods
- ❏ Manual wrapping C/C++ for Python

Running Python

- ❏ For Unix or Linux
 - ◆ \$ python
- ❏ Windows
 - ◆ C:> python
- ❏ Exit from Python
 - ◆ >>> quit()
- ❏ Running script file
 - ◆ \$ python script.py
 - ◆ or
 - ◆ \$./script.py
- ❏ Python file extensions
 - ◆ .py: Python source
 - ◆ .pyc: Python compiled byte code with most information
 - ◆ .pyo: Python compiled byte code optimized (-O), i.e., optimized .pyc
 - ◆ .whl: Python compressed format
- ❏ Command line syntax
 - ◆ \$ python [option] [-c cmd | -m mod | file | -] [args]
- ❏ Options
 - ◆ -d: It provides debug output.
 - ◆ -O: It generates optimized bytecode (resulting in .pyo files).
 - ◆ -S: Do not run import site to look for Python paths on startup.
 - ◆ -v: verbose output (detailed trace on import statements).
 - ◆ -X: disable class-based built-in exceptions (just use strings); obsolete starting with version 1.6.
 - ◆ -c cmd: run Python script sent in as cmd string
 - ◆ -m mod: importing module
 - ◆ file: run Python script from given file

Python

Comment

- ◆ Comments begin with the hash character ("#") and are terminated by the end of line.
- ◆ Python does not support comments that span more than one line.

Python is not 'free-format' language, but indentation using whitespace delimits program blocks.

- ◆ There are no block delimiters in Python. Instead, indentation does matter.
 - ◆ all the continuous lines indented with same number of spaces would form a block.
 - ◆ E.g., C language: {, }

Python is a case sensitive.

Python identifiers

- ◆ starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9).
- ◆ @, \$, and % are not allowed within identifier
- ◆ naming conventions
 - ◆ Class names start with an uppercase letter.
 - ◆ Identifier starting with '_' means private

Python

Multi-line statements

- ◆ use the line continuation character (\)

```
total = item_one + \
        item_two + \
        item_three
```

Multiple statements in a single line using semi-colon (;)

```
import sys; x = 'foo'; sys.stdout.write(x + '\n')
```

Statements contained within [], {}, or () brackets do not need to use \.

```
days = ['Monday', 'Tuesday', 'Wednesday',
        'Thursday', 'Friday']
```

Quotation

- ◆ single ('), double ("), and triple('"' or """) to denote string literals.
 - ◆ triple quotation: multi-line

```
word = 'word'
sentence = "This is a sentence."
paragraph = """This is a paragraph. It is
made up of multiple lines and sentences."""
```

```
print r'C:\nowhere' # results in C:\nowhere
print u'Hello' # results in 16-bit Unicode
print b'Hello' # results in 8-bit byte literal
```

Python reserved words

Keywords

```
and
as
assert
break
class
continue
def
del
elif
else
except
exec (changed to a built-in function in 3.x)
False (added in 3.x)
finally
for
from
global
if
import
in
is
```

_ (underscore) is the latest result
_ (underscore) used as don't care result

```
lambda
None (added in 3.x)
nonlocal (added in 3.x)
not
or
pass
print (changed to a built-in function in 3.x)
raise
return
True (added in 3.x)
try
while
with
yield
```

Copyright © 2016-2019 by Ando Ki

(21)

FUTURE
Design Systems

Python variable and data types

Python variables do not need explicit declaration to reserve memory space.

- ◆ The declaration happens automatically when you assign a value to a variable.
 - ◆ The equal sign (=) is used to assign values to variables.
- ◆ Variables can be deleted by using 'del'

```
var1 = 1; var2=10
print var1, var2
del var1, var2
```

Standard data types

- ◆ int: signed, 32-bits → 42
- ◆ long: signed, infinite, octal or hexadecimal → 42L, 0122L, 0xABCDEFL
- ◆ float: real value → 0.0, -21.9
- ◆ complex: complex number → 3.14j, -.65+0j, 10.9=7j
- ◆ str: string (immutable) → "Hello\x07\n"
- ◆ tuple: immutable group → 'World\x08\n'
- ◆ list: mutable group → (1,10,"what")
- ◆ dict: mutable group with key → [10,"1",9]
- {"one":1, "two":2, "name":'jone'}

Copyright © 2016-2019 by Ando Ki

(22)

FUTURE
Design Systems

Python strings and lists

String

- ◆ a contiguous set of characters represented in the quotation marks
- ◆ immutable
- ◆ index 0: starting
- ◆ slice operator: [] and [:]
- ◆ '+' operator: concatenation
- ◆ '*' operator: repetition

```
str = 'Hello World!'
```

```
print str          # Prints complete string
print str[0]       # Prints first character of the string
print str[2:5]     # Prints characters starting from 3rd to 5th
print str[2:]      # Prints string starting from 3rd character
print str * 2      # Prints string two times
print str + "TEST" # Prints concatenated string
```

List

- ◆ Ordered collection of data
- ◆ mutable
- ◆ A list contains items separated by commas and enclosed within square brackets ([]).
- ◆ Items of a list can be different data type

```
LST = [ ]
      [0] [1] [2] [3]
      [-4] [-3] [-2] [-1]
```

index

```
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tinylist = [123, 'john']
```

```
print len(list) # get the length of a list
print list      # Prints complete list
print list[0]   # Prints first element of the list
print list[1:3] # from 2nd till 3rd – list[1], list[2]
print list[2:]  # from 3rd element
print tinylist * 2 # Prints list two times
print list + tinylist # Prints concatenated lists
```

Copyright © 2016-2019 by Ando Ki

(23)

FUTURE
Design Systems

Python tuple and dictionary

Tuple

- ◆ Something like read-only list, i.e., immutable
 - ◆ ordered list of values.
- ◆ Items separated by comma and enclosed within parentheses (()).
- ◆ Need ',' to differentiate from the mathematical expression of number
 - ◆ $y=(2,)$ ← not number 2 but a tuple containing '2'.

```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )
tinytuple = (123, 'john')
```

```
print tuple      # Prints complete list
print tuple[0]   # Prints first element of the list
print tuple[1:3] # Prints elements starting from 2nd till 3rd
print tuple[2:]  # Prints elements starting from 3rd element
print tinytuple * 2 # Prints list two times
print tuple + tinytuple # Prints concatenated lists
```

Dictionary

- ◆ Something like hash table (lookup table) with key-value pairs in unordered fashion
 - ◆ ':' for key-value separation
 - ◆ duplicate keys are not allowed
 - ◆ duplicated value are just fine
- ◆ mutable
- ◆ items separated by commas and enclosed within curly braces {}.
- ◆ Items of a list can be different data type

```
dict = {}
dict['one'] = "This is one"
dict[2] = "This is two"
tinydict = {'name': 'john', 'code': 6734, 'dept': 'sales'}
```

```
print dict['one'] # Prints value for 'one' key
print dict[2]     # Prints value for 2 key
print tinydict    # Prints complete dictionary
print tinydict.keys() # Prints all the keys
print tinydict.values() # Prints all the values
```

Copyright © 2016-2019 by Ando Ki

(24)

FUTURE
Design Systems

Python set

Set

- ◆ Unordered collection of distinct elements
 - ◇ same element cannot be added
 - ◇ looks like dict without key
- ◆ Items separated by comma and enclosed within parentheses ({}).

```
animals = { 'cat', 'dog' }
print 'cat' in animals    # "True"
animals.add('fish')
print len(animals)        # "3" ← { 'cat', 'dog', 'fish' }
animals.add('cat')        # nothing happens since 'cat' exists
print len(animals)        # "3"
animals.remove('cat')
print len(animals)        # "2"
```

Python operator

Types of operator

No unary increment x++, decrement y--.

- ◆ Arithmetic Operators
 - ◇ +, -, *, /, %, **, //
 - ▢ **: exponent
 - ▢ //: floor division (get integer part of the result)
- ◆ Comparison (Relational) Operators
 - ◇ ==, !=, >, <, >=, <=
- ◆ Assignment Operators
 - ◇ =, +=, -=, *=, /=, %=, **=, //=
- ◆ Bitwise Operators
 - ◇ &, | ^, ~
- ◆ Logical Operators (not &&, ||)
 - ◇ 'and', 'or', 'not'
- ◆ Membership Operators
 - ◇ 'in', 'not in'
- ◆ Identity Operators
 - ◇ 'is'

```
a = 10
b = 20
list = [1, 2, 3, 4, 5];
```

```
if ( a in list ):
    print "Line 1 - a is available in the given list"
else:
    print "Line 1 - a is not available in the given list"
```

```
if ( b not in list ):
    print "Line 2 - b is not available in the given list"
else:
    print "Line 2 - b is available in the given list"
```

```
if ( a is b ):
    print "Line 1 - a and b have same identity"
else:
    print "Line 1 - a and b do not have same identity"
```

```
if ( id(a) == id(b) ):
    print "Line 2 - a and b have same identity"
else:
    print "Line 2 - a and b do not have same identity"
```

Python immutable (implicit sharing; i.e., copy on-write)

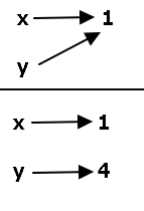
Python number is immutable.

- ◆ '1' is immutable
- ◆ 'y' after modification has different value.

```
x=1
y=x
y+=3

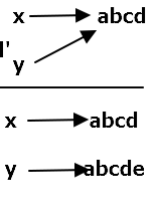
print "X ",x # it gives 4.5
print "Y ",y # it gives 7.5, not 4.5
```

```
>>> x = 1
>>> y = x
>>> y += 3
>>> x
1
>>> y
4
```



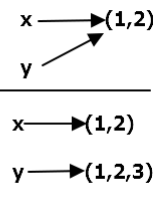
Python string is immutable.

```
>>> x = 'abcd'
>>> y = x
>>> y += 'e'
>>> x
abcd
>>> y
abcde
```



Python tuple is immutable

```
>>> x = (1,2)
>>> y = x
>>> y += (3,)
>>> x
(1,2)
>>> y
(1,2,3)
```



Primitive-like types are probably immutable. Container-like types are probably mutable.

Copyright © 2016-2019 by Ando Ki

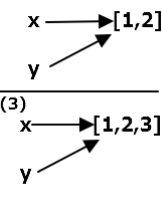
(27)

FUTURE
Design Systems

Python mutable

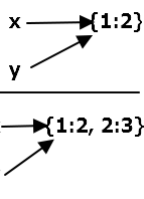
Python list is mutable.

```
>>> x = [1,2]
>>> y = x
>>> y.append(3)
>>> x
[1,2,3]
>>> y
[1,2,3]
```



Python dict and set are mutable.

```
>>> x = {1:2}
>>> y = x
>>> y[2] = 3
>>> x
{1:2, 2:3}
>>> y
{1:2, 2:3}
```



Copyright © 2016-2019 by Ando Ki

(28)

FUTURE
Design Systems

Python control flow

Control flow constructs (i.e., compound statements)

- ◆ if elif else
- ◆ for
 - ◆ else, break, continue, pass
- ◆ while
 - ◆ else, break, continue
- ◆ try except else

suite

- ◆ a group of individual statement making a single code block
- ◆ one or more lines following a colon (:) after header line
 - ◆ header line consists of statement (with the keyword) and terminated with a colon.

```
if cond1:
    instruct
    instruct
elif cond2:
    instruct
    instruct
else:
    instruct
    instruct
```

```
for var in set:
    instruct
    instruct
else:
    instruct
```

```
while cond:
    instruct
    instruct
else:
    instruct
```

```
try:
    instr
except exception:
    instruct when exception
else:
    instruct when no exception
```

```
# one-line if
if cond1: instruction
```

Copyright © 2016-2019 by Ando Ki

(29)

FUTURE
Design Systems

Python function

Function definition

- ◆ def
- ◆ first statement can be optional string
 - ◆ docstring
 - ▢ >>> print module.__doc__
 - ▢ >>> print module.function.__doc__
 - ▢ >>> print module.class.__doc__
 - ▢ >>> help(module)
 - ▢ >>> help(module.function)
 - ▢ >>> help(module.class)
- ◆ call-by-object argument passing
 - ◆ naming the object without copying
 - ◆ without assignment (=), it looks like call-by-referenc
 - ◆ with assignment (=), it looks like call-by-value
- ◆ use 'global' statement if required
- ◆ 'return expression' or 'return' or nothing

```
def new_function():
    instruct
```

```
def new_function(arg1):
    "documentation string"
    instruct
    instruct
```

```
def new_function(arg1, arg2):
    "documentation string"
    instruct
    instruct
    return var
```

```
def new_function(arg1, arg2):
    "documentation string"
    instruct
    instruct
    return var1, var2
```

Copyright © 2016-2019 by Ando Ki

(30)

FUTURE
Design Systems

Python function argument passing

Call-by-object

```
def func_arg(arg):  
    "argument passing"  
    arg.append(100)  
    print "first : ", arg  
    arg = [10, 20, 30]  
    print "second: ", arg  
    arg.append(40)  
    print "third : ", arg
```

```
var = [1, 2, 3]  
print var  
func_arg(var)  
print var
```

```
[1, 2, 3]  
first : [1, 2, 3, 100]  
second: [10, 20, 30]  
third : [10, 20, 30, 40]  
[1, 2, 3, 100]
```

Not 'arg' will be another object

call by value
call by reference
call by value
call by value
call by reference

Python function argument passing

Positional order

```
def func (arg1, arg 2):  
    print arg1, arg2  
  
func(1, 2)
```

Named passing

```
def func (arg1, arg 2):  
    print arg1, arg2  
  
func(arg2=2, arg1=1)
```

Default arguments

```
def func (arg1, arg 2=2):  
    print arg1, arg2  
  
func( arg1=1)
```

Variable-length arguments

```
def printinfo( arg1, *vartuple ):  
    "This prints a variable passed arguments"  
    print "Output is: "  
    print arg1  
    for var in vartuple:  
        print var  
    return;
```

```
# Now you can call printinfo function  
printinfo( 10 )  
printinfo( 70, 60, 50 )
```


Python modules

Module

- ◆ Python module is just a file with Python code

To import modules you use the filename without the .py extensions

- ◆ When imported, the module name is set to filename without .py extension even if it's renamed with ">>> import module as other_name."

Top-level statements will be executed once even if the file is imported several times even from different files

Python modules

Module importing

- ◆ 'import' statement enables to use Python source file[s]

- module is loaded only once, regardless of the number of times it is imported.

- ◆ >>> import module1[, module2[, ... moduleN]]
 - This looks for 'module1.py' in the search path
 - ◆ current directory; PYTHONPATH; /usr/local/lib/python
 - ◆ >>> import sys
 - ◆ >>> print sys.path

- ◆ Now use any functions in the module as follows

- >>> module1.func(x, y)

- ◆ Module under a specific directory

- ◆ >>> import sys
 - ◆ >>> sys.path.insert(0, 'directory')
 - ◆ >>> import module

| | |
|------------------------|--|
| import mymodule | Brings all elements of mymodule in, but must refer to as mymodule.<elem> |
| import mymodule as my | Brings all elements of mymodule in, but must refer to as my.<elem> |
| from mymodule import x | Imports x from mymodule right into this namespace |
| from mymodule import * | Imports all elements of mymodule into this namespace - No need of mymodule to refer element of it |

Import only specific attributes from a module into the current namespace

- ◆ >>> from module1 import my_func
- ◆ >>> from module2 import * # import all names

Python `__name__` and `__main__`

When Python reads a source file,

1. sets a few special variables including `__name__`, and then
2. executes all of the code found in the file.

'`__name__`' will be '`__main__`' when the module (the source file) as the main program.

```
#my_modA.py
print(__file__)
print(__name__)
if __name__ == '__main__':
    print("Test code here for ", __file__)
```

```
#my_modB.py
import my_modA
print(__file__)
print(__name__)
if __name__ == '__main__':
    print("Test code here for ", __file__)
```

```
Terminal File Edit View Search Terminal Help
[adki@AndoUbuntu] ls
my_modA.py* my_modB.py*
[adki@AndoUbuntu] python my_modA.py
my_modA.py
__main__
('Test code here for ', 'my_modA.py')
[adki@AndoUbuntu]
```

Note that `__name__` will be `__main__`.

```
adki@AndoUbuntu: ~/work/projects/Python/python-projects/name
[adki@AndoUbuntu] ls
my_modA.py* my_modB.py*
[adki@AndoUbuntu] python my_modB.py
/home/adki/work/projects/Python/python-projects/name/my_modA.py
my_modA
my_modB.py
__main__
('Test code here for ', 'my_modB.py')
[adki@AndoUbuntu]
```

Note that `__name__` of 'my_modA.py' will not be `__main__`.

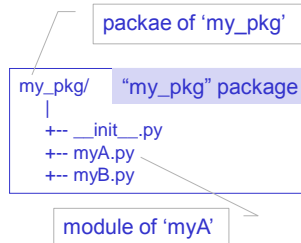
Python package

Package is a namespace for a collection of modules

Package is the minimal unit of code distribution in Python

Making a package

1. make a directory
2. put all modules (i.e., files) to the directory
3. create an empty `__init__.py` file under the directory



Any directory with `__init__.py` by convention is a Python package

- Additional purpose of '`__init__.py`' is initialization
- When you import a package, the `__init__.py` module of the package is executed

To use module in the package

- package is a namespace for modules, so you don't import the package itself, you import a module from a package

```
>>> import my_pkg.myA
>>> from my_pkg import myA
>>> from my_pkg import *
```

import module from package

Star import

Python search path

Python order of searching

- ◆ 1. current directory
- ◆ 2. 'sys.path'
 - ◆ Usually it contains the current directory
- ◆ 3. 'PYTHONPATH' environment variable

Python has module search path available at runtime as sys.path.

- ◆ If you run a module as a script file, the containing directory is added to sys.path, otherwise, the current directory is added to it

Python command line arguments

Python 'sys' module provides access to any command-line arguments via the 'sys.argv'

There is no 'argc' with Python.

How to get argc using argv

- ◆ >>> import sys
- ◆ >>> len(sys.argv)
- ◆ >>> sys.argv

```
Terminal File Edit View Search Terminal Help
[adki@AndoUbuntu]
[adki@AndoUbuntu] cat argc_argv.py
import sys

def main(argc, argv):
    print('main ' + str(argc) + ", " + str(argv))
    for v in argv:
        print (argv.index(v), v)

if __name__ == '__main__':
    main(len(sys.argv), sys.argv)
[adki@AndoUbuntu]
[adki@AndoUbuntu]
[adki@AndoUbuntu] python argc_argv.py A B C
main 4, ['argc_argv.py', 'A', 'B', 'C']
(0, 'argc_argv.py')
(1, 'A')
(2, 'B')
(3, 'C')
[adki@AndoUbuntu]
```

Python output and input

Printing to the screen

A newline appended by default

◆ >>> print "....."

◆ >>> print ".....",

◆ >>> print "....."

Print without newline and useful in the Python script

Reading keyboard input

◆ raw_input: reads one line from standard input and returns it as a string (removing the trailing newline)

◆ >>> str = raw_input("prompt or your command if any")

◆ input: similar to 'raw_input', except assuming the input is a valid Python expression and returns the evaluated result to you.

◆ >>> str = input("prompt")

Python regular expression

Regular expression to match or find a pattern in the string

◆ match: checks for a match only at the beginning of the string

◆ search: searches for first occurrence of the pattern

◆ search & replace:

◆ >>> re.match(pattern, string, flags=0)

◆ >>> re.search(pattern, string, flags=0)

◆ >>> re.sub(pattern, repl, string, max=0)

Python modules

- ❏ 'dis' module
 - ◆ Python disassembler
 - ◆ > import dis
- ❏ 'pdb' module
 - ◆ Python debugger
 - ◆ > import pdb
- ❏ 'profile' module
 - ◆ Python profiler
 - ◆ \$ cProfile.py script.py
- ❏ 'tabnanny' module
 - ◆ ambiguous indentation
 - ◆ \$ tabnanny.py -v script.py
- ❏ 'math', 'sys', 're', 'os', 'os.path', 'logging', 'collections', 'struct', 'decimal', 'datetime', 'time', 'tempfile', 'random', 'shutil', 'glob', 'subprocess', ...
- ❏ 'os' module
 - ◆ system functions
- ❏ 'CTypes' module
 - ◆ calling the functions of dlls/shared libraries
- ❏ SciPy package
 - ◆ 'NumPy' module
 - ◆ N-dim array
 - ◆ 'SciPy' module
 - ◆ Scientific computing
- ❏ 'matplotlib'
 - ◆ numerical plotting

Python modules

- ❏ 'PIL/Pillow'
- ❏ 'IPython'
 - ◆ Interactive Python
- ❏ Modules for TensorFlow
 - ◆ 'numpy'
 - ◆ a numerical processing package
 - ◆ 'dev'
 - ◆ enables adding extensions to Python
 - ◆ 'pip'
 - ◆ to install and manage certain Python packages
 - ◆ 'wheel'
 - ◆ manage Python compressed packages in the wheel (.whl) format

Torch and PyTorch

* A tensor library like Numpy with strong GPU support.

References

Python tutorial

- ◆ <http://www.tutorialspoint.com/python/>
- ◆ <http://www-h.eng.cam.ac.uk/help/languages/python/>