INTRODUCTORY COURSE TO PRACTICAL PYTHON PROGRAMMING

Engku Ahmad Faisal

eafaisal@gmail.com

github.com/efaisal

twitter.com/efaisal

facebook.com/eafaisal

plus.google.com/100085338667833364551

ABOUT THIS COURSE

- Fundamental course on programming and Python
- 3 days
- Format:
  - Lectures
  - Hands-on exercises
  - Handout
- Feel free to ask question anytime
- Always use the official Python documentation (https://docs.python.org/2/ or https://docs.python.org/3/)

INTRODUCING PROGRAMMING

What is Programming?

- Instruction to do something to achieve a desired result
- Using series of steps or actions that a programmer defines (procedure)
- Each action works on certain object(s) or data in order to attain a well defined goal

```
334             if dot == '':
335                 base = suffix
336
337             yield dir + base
338
339     def resolve(self, makefile, va
340         util.joiniter(fd, self.bas
    …iables, setting)))
341
342 class AddSuffixFunction(Function):
343     name = 'addprefix'
344     minargs = 2
345     maxargs = 2
346
347     __slots__ = Function.__slots__
348
349     def resolve(self, makefile, va
350         suffix = self._arguments[0
351
352         fd.write(' '.join([w + suf
    …e, variables, setting)]))
353
354 class AddPrefixFunction(Function):
355     name = 'addsuffix'
356     minargs = 2
357     maxargs = 2
358
359     def resolve(self, makefile, va
360         prefix = self._arguments[0
361
362         fd.write(' '.join([prefix
    …e, variables, setting)]))
363
364 class JoinFunction(Function):
365     name = 'join'
366     minargs = 2
367     maxargs = 2
```

REAL WORLD EXAMPLE OF PROGRAMMING

Shampoo Instruction

- Wash
- Rinse
- Repeat

Do you notice the problem with the instruction?

COMPUTER PROGRAMMING

- Computer is a dumb device that cannot understand human language
- You, as a programmer, need to tell the computer precisely what to do
- Use programming language which helps to translate your instruction so that computer can understand

WHAT IS PYTHON?

An Overview

- A high level, general purpose programming language
- High level: don't need to know the details of hardware
- General purpose: can be used to build many kind of applications (vs domain specific language such as HTML, SQL, MATLAB)

WHAT IS PYTHON? (cont)

A little history:

- Designed by Guido van Rossum
- First release in 1991
- Has many implementation: CPython, PyPy, Jython, IronPython
- We'll be using the reference implementation: CPython

WHAT IS PYTHON? (cont)

Why learn Python?

- Low learning curve
- “Enforce” good programming practice
- Multi-platforms: Windows, Linux, Mac
- “Batteries” included

WHAT IS PYTHON? (cont)

Examples of applications that can be built using Python:

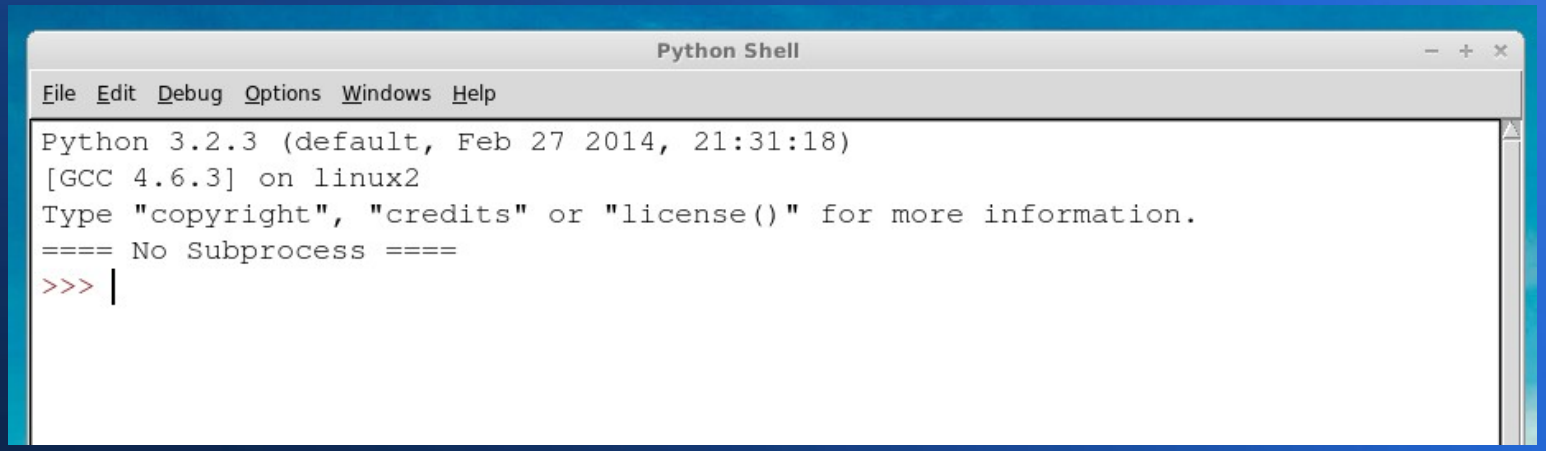
- Desktop applications/games
- Web-based applications
- System administrator tasks automation
- Scientific & engineering data analysis

WHAT IS PYTHON? (cont)

Who uses Python?

- Giant companies & government agencies:
  - Google, Dropbox, Instagram
  - CERN, NASA
- In Malaysia:
  - TUDM, INCEIF, Star Newspaper
- You!
  Download from www.python.org

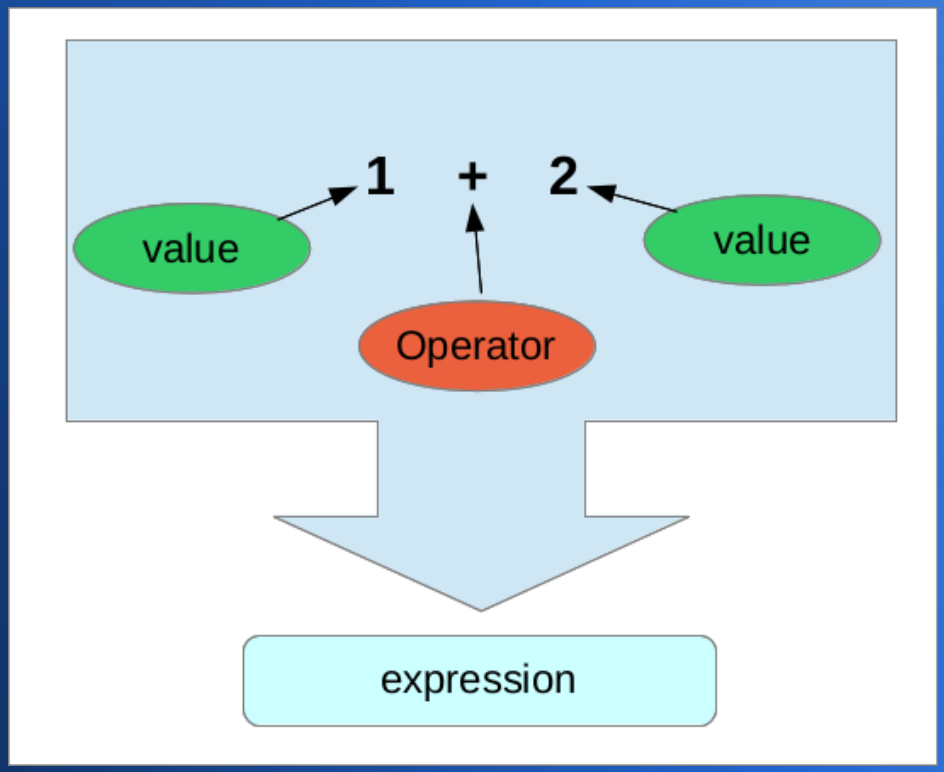
INTRODUCING IDLE



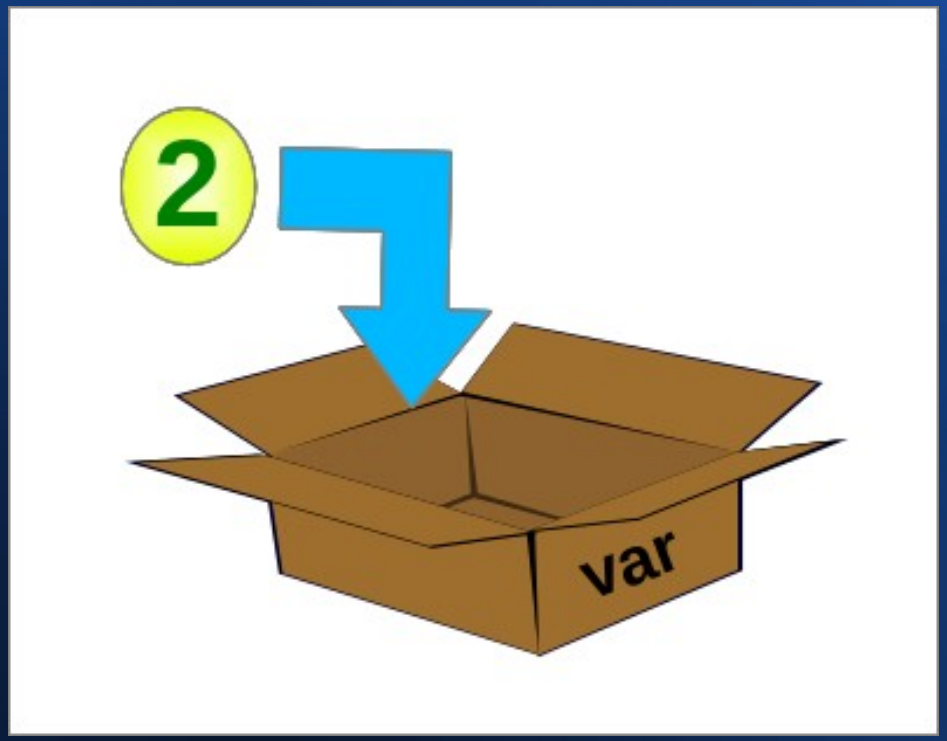
- IDLE is an integrated development environment
- Part of Python standard library using Tkinter as GUI toolkit
- Provides Python shell and a text editor
- Not great but good enough for our purpose

VALUE & EXPRESSION

- Type on the Python shell:
  >>> 1 + 2
- As expected the result is 3
- The number 1 and 2 are called values
- The symbol + is an operator
- “1 + 2” together forms an expression



STORING VALUE



- Store into variable
- Using the = sign
- Examples:

```
>>> var = 2
>>> var
2
>>> a = 1
>>> var + a
3
>>> result = var + a
>>> result
3
```

DATATYPES: NUMBERS

- Integers

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

- Floats

```
>>> b = 2.5
>>> type(b)
<class 'float'>
>>>
```

- Fractions

```
>>> import fractions
>>> # Simulate 1/3
>>> c = fractions.Fraction(1, 3)
>>> c
Fraction(1, 3)
>>>
```

- Complex numbers

DATATYPES: NUMBERS (cont)

Common Operators for Numbers			
+	Addition	>	Greater than
-	Substraction	>=	Greater than or equal to
*	Multiplication	<	Lesser than
/	Division	<=	Lesser than or equal to
%	Modulus (calculate remainder)		

PRIMITIVE DATATYPES

DATATYPES: STRINGS

- Enclosed by single quotes

```
>>> 'this is a string'
'this is a string'
>>>
```

- Enclosed by double quotes

```
>>> “this is also a string”
'this is also a string'
>>>
```

- Enclosed by triple single or double quotes for multilines string

```
>>> '''first multi
… line string'''
'first multi\nline string'
>>> “””second multi
… line string”””
'second multi\nline string'
>>>
```

DATATYPES: STRINGS

- Concatenate
  ```
  >>> a = 'my '
  >>> a + 'string'
  'my string'
  >>>
  ```

- Repeat
  ```
  >>> b = 'ha '
  >>> b * 3
  'ha ha ha '
  >>>
  ```

- Long string
  ```
  >>> mystr = ('put in bracket '
  … 'to handle long string')
  >>> mystr
  'put in bracket to handle … '
  ```

DATATYPES: STRING

- String is a sequence of characters & can be indexed/subscripted

```
>>> mystr = 'Python'
>>> mystr[0]
'P'
>>> mystr[5]
'n'
>>> mystr[-1]
'n'
>>> mystr[-6]
'P'
```

DATATYPE: STRINGS

- String is a sequence of characters & can be sliced

```
>>> mystr = 'Python'
>>> mystr[1:]
'ython'
>>> mystr[1:5]
'ytho'
>>>
```

- To get the length of characters, use the built-in function len()

```
>>> len(mystr)
6
>>>
```

BASIC
DATA STRUCTURE

DATA STRUCTURE: LIST

- Mutable (changeable), compound (group of values) data type
- Blank lists
  >>> a = []
  >>> b = list()
- Lists with default values
  >>> a = [1, 2, 3, 4]
  >>> b = ['x', 'y', 'z']
- Look at the available methods to manipulate list
- List comprehension: a more advance topic on list will be covered later

DATA STRUCTURE: TUPLE

- Immutable (unchangeable), compound (group of values) data type
- Blank tuples
  >>> a = ()
  >>> b = tuple()
- Tuples with values
  >>> a = (1, 2, 3, 4)
  >>> b = ('x', 'y', 'z')
- Look at the available methods for tuple datatype

DATA STRUCTURE: DICTIONARY

- Mapping datatype (key-value); similar to associative array in some languages

- Blank dictionaries
  ```
  >>> a = {}
  >>> b = dict()
  ```

- Dictionaries with values
  ```
  >>> a = {'a': 1, 'b': 2}
  >>> b = dict((('a', 1), ('b', 2)))
  ```

CONTROL FLOWS

CONTROL FLOW: IF

- Conditional control flow
- Example (Note: identation is 4 spaces):

```
>>> x = 5
>>> if x < 0:
...     print('Negative value')
... elif x == 0:
...     print('Zero')
... elif x > 0 and x < 11:
...     print('Between 1 and 10')
... else:
...     print('More than 10')
'Between 1 and 10'
>>>
```

CONTROL FLOW: FOR

- Iterates over items in iterables (e.g. list, tuple)
- Example:

```
>>> for n in range(2, 10):
…     for x in range(2, n):
…         if n % x == 0:
…             break
…     else:
…         print(n, 'is a prime number') # Python 3
…         print n, 'is a prime number') # Python 2
```

- *break* is used to terminate iteration
- *else* is executed if no *break* occurs in the *for* iteration

CONTROL FLOW: FOR

- Another example:

```
>>> for i in range(2, 10):
…     if i % 2 == 0:
…         print(i, ' is an even number') # Python 3
…         print i, ' is an even number'  # Python 2
…         continue
…     print(i, ' is an odd number') # Python 3
…     print i, ' is an odd number'  # Python 2
```

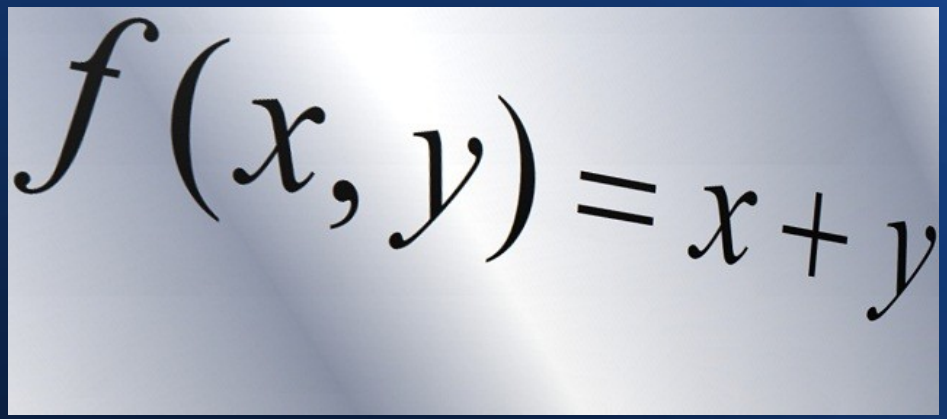
- *continue* keyword means continues with next iteration

CONTROL FLOW: WHILE

- Used to loop or repeat a section of code for as long as the condition (expression) defined in the statement is true
- Example, to loop forever:
  ```
  >>> while True:
  …     pass
  ```
- *pass* keyword means “do no action” or “do nothing”

FUNCTIONS

WHAT IS A FUNCTION?



- A named section of a program that performs specific task
- Allow to organize code block which repeatedly used into a reusable procedure or routine
- Python comes with a lot of built-in functions which we already seen some of them

FUNCTION: DEFINING

- Using the keyword *def*
- Followed by a name, patentheses (for possible arguments) and a colon
- Body of a function must be indented
- Variables defined in a function is stored in the local symbol table
- May return a value using *return* keyword

- Example: Fibonacci series

```
>>> def fib(n):
...     """Doc string"""
...     a, b = 0, 1
...     while a < n:
...         print a,                # Py2
...         print(a, end=' ') # Py3
...         a, b = b, a+b
...     print()                     # Py3
...
>>> fib(2000)
```

FUNCTION: DEFAULT ARGUMENT

- Sometimes it's useful to define default value to argument of a function
- Example:
  >>> def calculate(arg, val=1):
  …
- Default value is evaluated at the point of function definition and evaluated on once
- Example:
  >>> i = 2
  >>> def f(arg=i):
  …
  >>> i = 5

FUNCTION: KEYWORD ARGUMENT

- Function can be called using keyword argument
- Example:

```
>>> def func(arg1=1, say='hello', arg3=5):
...     print(say)
...     print(arg1)
...     print(arg3)
>>> func(say='world') # Using keyword argument
>>> func(1, 'world')     # Using positional arguments
>>> func()               # Using default arguments
>>>
```

FUNCTION: ARBITRARY ARGUMENT

- Though seldomly used, but handy when defining a function without knowing the possible argument upfront
- Example:

```
>>> def myfunc(name='test', *args, **kwargs):
…     print(name)
…     print(args)
…     print(kwargs)
…
>>> myfunc()
>>> myfunc('test2', 1, 2, 3)
>>> myfunc('test3', 1, 2, arg3='hello', arg4='world')
```

FUNCTION: ANONYMOUS

- Anonymous function can be created using the keyword *lambda*
- Example:

```
>>> is_even = lambda n: False if n == 0 or n % 2 else True
...
>>> is_even(0)
>>> is_even(1)
>>> is_even(2)
>>>
```

- Typically used when you need a function object

OBJECT ORIENTED PROGRAMMING (OOP)

OOP: INTRODUCTION

- Programming paradigm based on the concept of data structure revolving around objects rather than actions
- Object contain attributes (variable within the scope of the object) and methods (functions within the object)
- The definition of object data structure is by using classes
- In Python, everything is an object
- Example:
  >>> a = 1
  …
  >>> type(a)

OOP: DEFINING A CLASS

- Using *class* keyword
- Example:

```
>>> class MyClass:
…     def __init__(self, say='what'):
…         self.say = say
…     def shout(self):
…         print(self.say)
…
>>> a = MyClass('Hello, world!')
>>> a.shout()
>>> a.say = 'What say you?'
>>> a.shout()
```

OOP: ARBITRARY DATA STRUCTURE

- Python allow you to create a new data structure
- Example:

```
>>> class Point:
…     def __init__(self, x, y):
…         self.x, self.y = x, y
…     def __add__(self, pt):
…         return Point(self.x+pt.x, self.y+pt.y)
…     def __sub__(self, pt):
…         return Point(self.x-pt.x, self.y-pt.y)
…     def __eq__(self, pt):
…         return self.x == pt.x and self.y == pt.y
```

OOP: INHERITANCE

- Derive a class from a base class
- Example:

```
>>> class Tyre:
…     def __init__(self, n=4):
…         self.numtyre = num
…
>>> class Color:
…     def __init__(self, c='red'):
…         self.color = c
…
```

- Example (cont):

```
>>> class Car(Tyre):
…     def __init__(self):
…         super(Car, self).__init__()
…
>>> a = Car()
>>> a.numtyre
```

- Show multiple inheritance quirks

ERRORS & EXCEPTIONS

ERRORS & EXCEPTIONS

- Two type of error messages
  - Syntax error
  - Exceptions
- Syntax error example:
  >>> if a ok
- Exceptions examples:
  >>> z
  >>> 4 + “1”
- Exception happens when the syntax is correct during execution but error is detected

EXCEPTION HANDLING

- Unlike syntax error, exception can be handle by your code
- Example:

```
>>> try:
...     z / 0
... except NameError:
...     print(“Don't know what z is!!”)
... except ZeroDivisionError:
...     print(“Can't divide z with 0!”)
... except:
...     print(“Some other error”)
...
>>>
```

EXCEPTION HANDLING (CONT)

- Another example:

```
>>> try:
…      z / 0
… except (NameError, ZeroDivisionError) as e:
…     print(“Error: {0}.format(e.args[0]))
… except:
…      print('some other error')
…
```

- There is also try-except-else and try-except-else-finally statement
- The *else* block will be evaluated after successful run of *try* block while the *finally* block will always executed

RAISING EXCEPTION

- Exception can be manually raised using the keyword *raise*
- Example:
  >>> raise NameError('duh! Var is not defined')
- Another example using custom error:
  >>> class MyCustomError(Exception):
  … pass
  …
  >>> raise MyCustomError('wow! my own error')

MODULES

MODULE: INTRODUCTION

- A module is a file (with .py extension) containing Python definitions and statements.
- The file name will become the module name and can be imported from your main program
- Technically, your main program is also a (special) module called “__main__” module.
- Python searches for modules in the current directory follows by the path defined in PYTHONPATH then in installation dependent default.
- Python comes with a lot of standard modules

MODULE: EXAMPLE

- Create a file called mymod.py (this will be your module)

```
def func1():
    print ('func1 from mymod')

def func2():
    print('func2 from mymod')
```

- Open Python shell from the same directory

```
>>> import mymod
>>> mymod.func1()
>>> mymod.func2()
```

- Module is accessed using dot notation

MODULE: PACKAGE

- Package is a way to structuring or organising modules into a more logical collection as you see fit

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

BATTERIES INCLUDED

BATTERIES INCLUDED

- Standard installation of Python comes with lots of built-in functions and an array of standard libraries.
- In some cases, these functions and libraries are all you need to build your application.
- These functions and libraries in most cases are well documented.
- While there are a lot to cover, we'll only walkthrough a subset.
- The focus for this section will be the ones you'll (probably) often use.

BUILT-IN FUNCTIONS

- enumerate([seq, start=0])
- raw_input([prompt])
- len(s)
- range([start,] stop[, step]), xrange()
- open(name[, mode])
- set([iterable])
- int(), oct(), hex(), bin()

STANDARD LIBRARIES

- sys
- os
- time
- datetime
- math
- decimal
- logging
- doctest & unittest

INTRODUCTORY COURSE TO PRACTICAL PYTHON PROGRAMMING

Engku Ahmad Faisal

eafaisal@gmail.com

github.com/efaisal

twitter.com/efaisal

facebook.com/eafaisal

plus.google.com/100085338667833364551

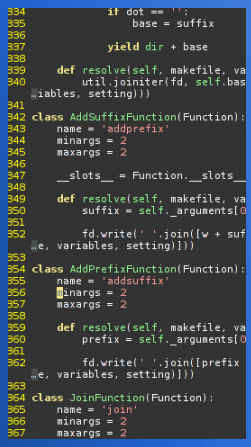
ABOUT THIS COURSE

- Fundamental course on programming and Python
- 3 days
- Format:
  - Lectures
  - Hands-on exercises
  - Handout
- Feel free to ask question anytime
- Always use the official Python documentation (https://docs.python.org/2/ or https://docs.python.org/3/)

INTRODUCING PROGRAMMING

What is Programming?

- Instruction to do something to achieve a desired result
- Using series of steps or actions that a programmer defines (procedure)
- Each action works on certain object(s) or data in order to attain a well defined goal



```
334                 if dot == '':
335                     base = suffix
336
337                 yield dir + base
338
339         def resolve(self, makefile, va
340             util.joiniter(fd, self.bas
    iables, setting)))
341
342 class AddSuffixFunction(Function):
343     name = 'addprefix'
344     minargs = 2
345     maxargs = 2
346
347     __slots__ = Function.__slots__
348
349     def resolve(self, makefile, va
350         suffix = self._arguments[0
351
352         fd.write(' '.join([w + suf
    e, variables, setting)]))
353
354 class AddPrefixFunction(Function):
355     name = 'addsuffix'
356     minargs = 2
357     maxargs = 2
358
359     def resolve(self, makefile, va
360         prefix = self._arguments[0
361
362         fd.write(' '.join([prefix
    e, variables, setting)]))
363
364 class JoinFunction(Function):
365     name = 'join'
366     minargs = 2
367     maxargs = 2
```

REAL WORLD EXAMPLE OF PROGRAMMING

Shampoo Instruction

- Wash
- Rinse
- Repeat

Do you notice the problem with the instruction?

COMPUTER PROGRAMMING

- Computer is a dumb device that cannot understand human language
- You, as a programmer, need to tell the computer precisely what to do
- Use programming language which helps to translate your instruction so that computer can understand

WHAT IS PYTHON?

An Overview

- A high level, general purpose programming language
- High level: don't need to know the details of hardware
- General purpose: can be used to build many kind of applications (vs domain specific language such as HTML, SQL, MATLAB)

WHAT IS PYTHON? (cont)

A little history:

- Designed by Guido van Rossum
- First release in 1991
- Has many implementation: CPython, PyPy, Jython, IronPython
- We'll be using the reference implementation: CPython

WHAT IS PYTHON? (cont)

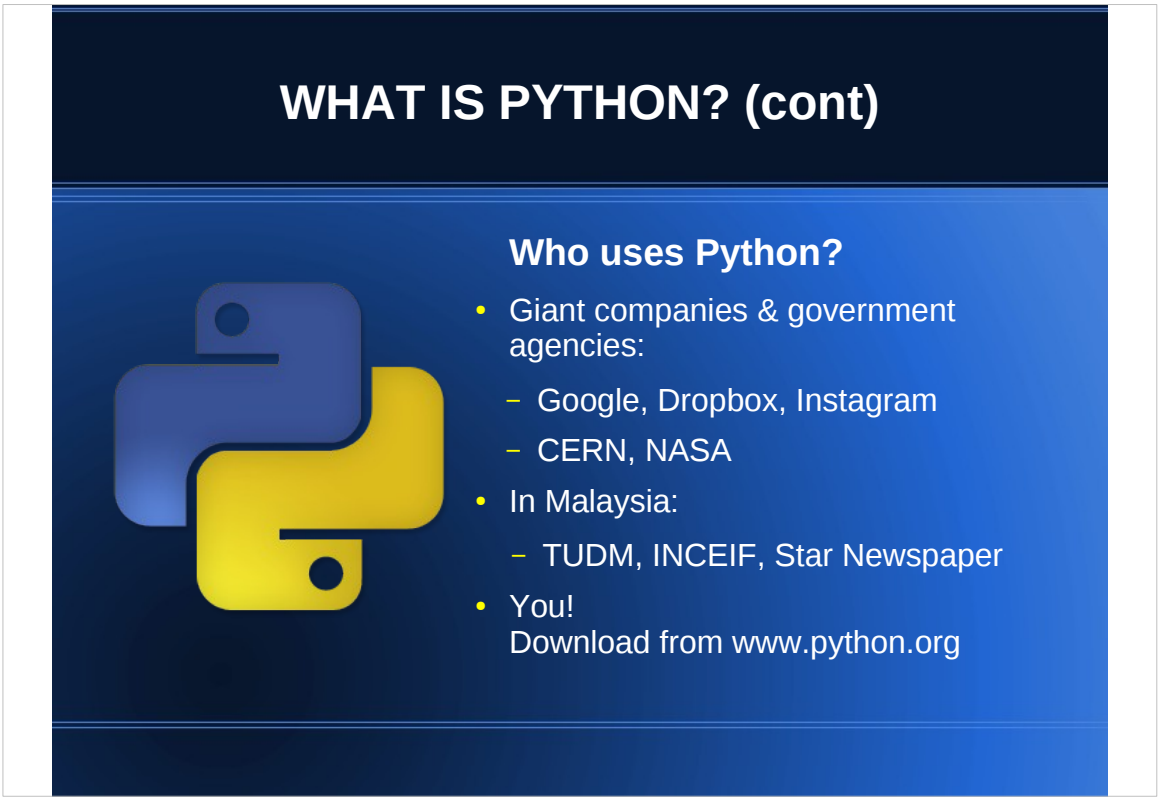
Why learn Python?

- Low learning curve
- “Enforce” good programming practice
- Multi-platforms: Windows, Linux, Mac
- “Batteries” included

WHAT IS PYTHON? (cont)

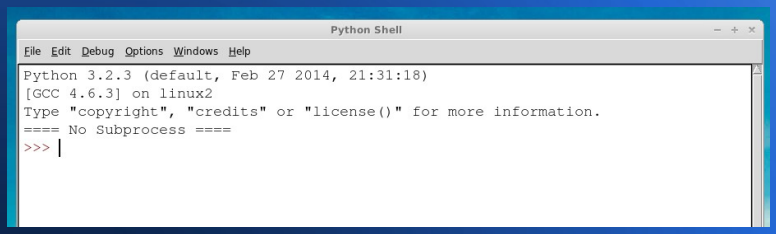
Examples of applications that can be built using Python:

- Desktop applications/games
- Web-based applications
- System administrator tasks automation
- Scientific & engineering data analysis



Show demo:
1. Game
2. Facial recognition

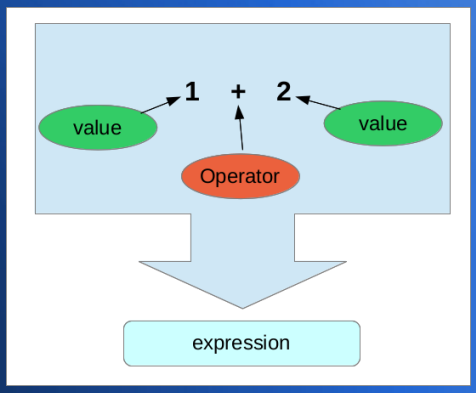
INTRODUCING IDLE



- IDLE is an integrated development environment
- Part of Python standard library using Tkinter as GUI toolkit
- Provides Python shell and a text editor
- Not great but good enough for our purpose

VALUE & EXPRESSION

- Type on the Python shell:
  >>> 1 + 2
- As expected the result is 3
- The number 1 and 2 are called values
- The symbol + is an operator
- “1 + 2” together forms an expression



STORING VALUE

- Store into variable
- Using the = sign
- Examples:

```
>>> var = 2
>>> var
2
>>> a = 1
>>> var + a
3
>>> result = var + a
>>> result
3
```

DATATYPES: NUMBERS

- Integers

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

- Floats

```
>>> b = 2.5
>>> type(b)
<class 'float'>
>>>
```

- Fractions

```
>>> import fractions
>>> # Simulate 1/3
>>> c = fractions.Fraction(1, 3)
>>> c
Fraction(1, 3)
>>>
```

- Complex numbers

DATATYPES: NUMBERS (cont)

Common Operators for Numbers			
+	Addition	>	Greater than
-	Substraction	>=	Greater than or equal to
*	Multiplication	<	Lesser than
/	Division	<=	Lesser than or equal to
%	Modulus (calculate remainder)		

PRIMITIVE DATATYPES

DATATYPES: STRINGS

- Enclosed by single quotes

```
>>> 'this is a string'
'this is a string'
>>>
```

- Enclosed by double quotes

```
>>> “this is also a string”
'this is also a string'
>>>
```

- Enclosed by triple single or double quotes for multilines string

```
>>> '''first multi
… line string'''
'first multi\nline string'
>>> “””second multi
… line string”””
'second multi\nline string'
>>>
```

DATATYPES: STRINGS

- Concatenate
```
>>> a = 'my '
>>> a + 'string'
'my string'
>>>
```

- Repeat
```
>>> b = 'ha '
>>> b * 3
'ha ha ha '
>>>
```

- Long string
```
>>> mystr = ('put in bracket '
… 'to handle long string')
>>> mystr
'put in bracket to handle … '
```

DATATYPES: STRING

- String is a sequence of characters & can be indexed/subscripted

```
>>> mystr = 'Python'
>>> mystr[0]
'P'
>>> mystr[5]
'n'
>>> mystr[-1]
'n'
>>> mystr[-6]
'P'
```

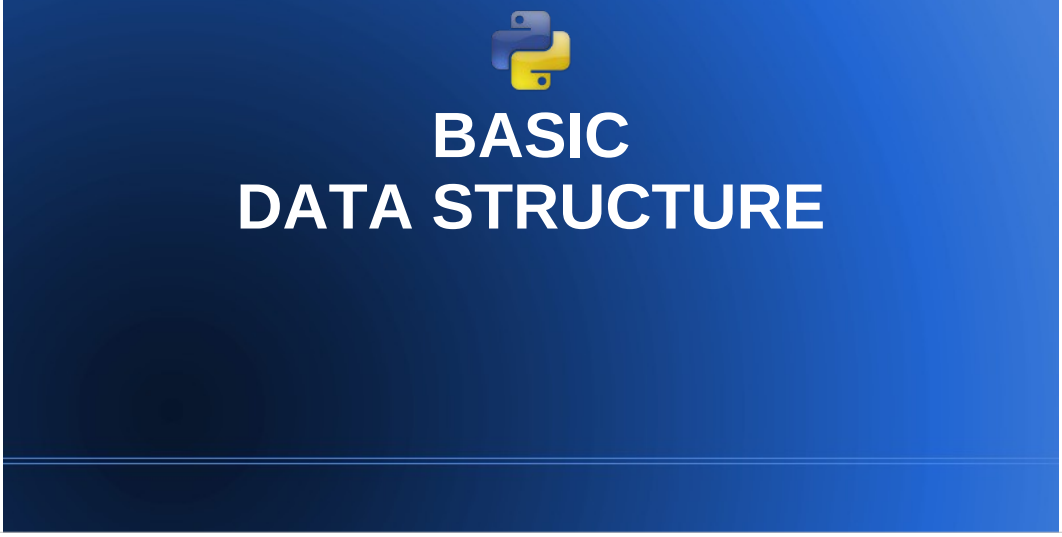
DATATYPE: STRINGS

- String is a sequence of characters & can be sliced

```
>>> mystr = 'Python'
>>> mystr[1:]
'ython'
>>> mystr[1:5]
'ytho'
>>>
```

- To get the length of characters, use the built-in function len()

```
>>> len(mystr)
6
>>>
```


BASIC
DATA STRUCTURE

DATA STRUCTURE: LIST

- Mutable (changeable), compound (group of values) data type
- Blank lists
  >>> a = []
  >>> b = list()
- Lists with default values
  >>> a = [1, 2, 3, 4]
  >>> b = ['x', 'y', 'z']
- Look at the available methods to manipulate list
- List comprehension: a more advance topic on list will be covered later

DATA STRUCTURE: TUPLE

- Immutable (unchangeable), compound (group of values) data type
- Blank tuples
  >>> a = ()
  >>> b = tuple()
- Tuples with values
  >>> a = (1, 2, 3, 4)
  >>> b = ('x', 'y', 'z')
- Look at the available methods for tuple datatype

DATA STRUCTURE: DICTIONARY

- Mapping datatype (key-value); similar to associative array in some languages
- Blank dictionaries
  ```
  >>> a = {}
  >>> b = dict()
  ```
- Dictionaries with values
  ```
  >>> a = {'a': 1, 'b': 2}
  >>> b = dict((('a', 1), ('b', 2)))
  ```


CONTROL
FLOWS

CONTROL FLOW: IF

- Conditional control flow
- Example (Note: identation is 4 spaces):

```
>>> x = 5
>>> if x < 0:
...     print('Negative value')
... elif x == 0:
...     print('Zero')
... elif x > 0 and x < 11:
...     print('Between 1 and 10')
... else:
...     print('More than 10')
'Between 1 and 10'
>>>
```

CONTROL FLOW: FOR

- Iterates over items in iterables (e.g. list, tuple)
- Example:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             break
...     else:
...         print(n, 'is a prime number') # Python 3
...         print n, 'is a prime number') # Python 2
```

- *break* is used to terminate iteration
- *else* is executed if no *break* occurs in the *for* iteration

CONTROL FLOW: FOR

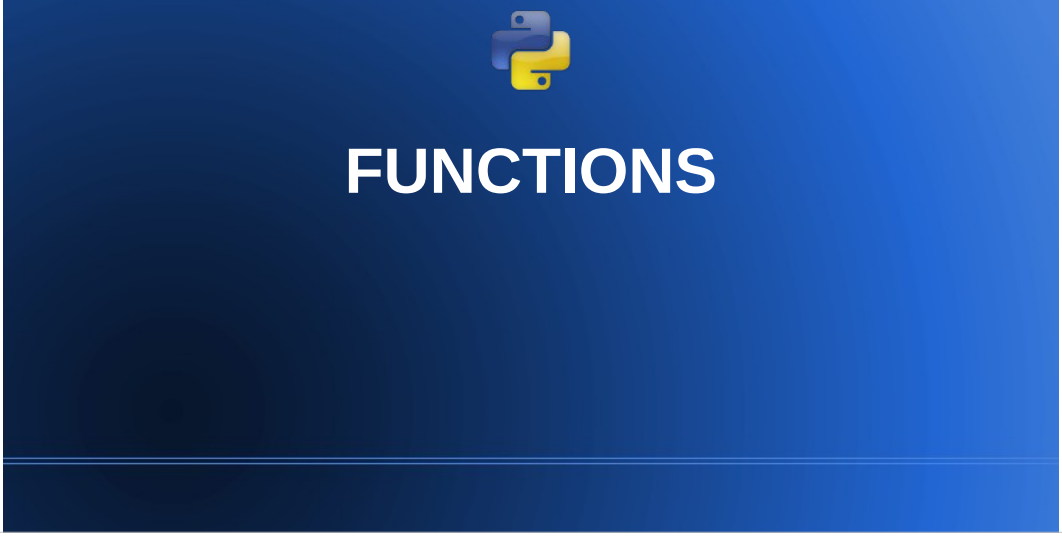
- Another example:

```
>>> for i in range(2, 10):
…     if i % 2 == 0:
…         print(i, ' is an even number') # Python 3
…         print i, ' is an even number'  # Python 2
…         continue
…     print(i, ' is an odd number') # Python 3
…     print i, ' is an odd number'  # Python 2
```

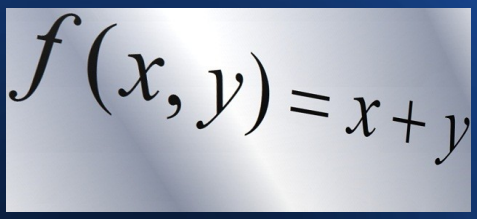
- *continue* keyword means continues with next iteration

CONTROL FLOW: WHILE

- Used to loop or repeat a section of code for as long as the condition (expression) defined in the statement is true
- Example, to loop forever:
  >>> while True:
  … pass
- *pass* keyword means “do no action” or “do nothing”


FUNCTIONS

WHAT IS A FUNCTION?



- A named section of a program that performs specific task
- Allow to organize code block which repeatedly used into a reusable procedure or routine
- Python comes with a lot of built-in functions which we already seen some of them

FUNCTION: DEFINING

- Using the keyword *def*
- Followed by a name, patentheses (for possible arguments) and a colon
- Body of a function must be indented
- Variables defined in a function is stored in the local symbol table
- May return a value using *return* keyword

- Example: Fibonacci series

```
>>> def fib(n):
…     """Doc string"""
…     a, b = 0, 1
…     while a < n:
…         print a,              # Py2
…         print(a, end=' ') # Py3
…         a, b = b, a+b
…     print()                  # Py3
…
>>> fib(2000)
```

FUNCTION: DEFAULT ARGUMENT

- Sometimes it's useful to define default value to argument of a function
- Example:
  >>> def calculate(arg, val=1):
  …
- Default value is evaluated at the point of function definition and evaluated on once
- Example:
  >>> i = 2
  >>> def f(arg=i):
  …
  >>> i = 5

FUNCTION: KEYWORD ARGUMENT

- Function can be called using keyword argument
- Example:

```
>>> def func(arg1=1, say='hello', arg3=5):
...     print(say)
...     print(arg1)
...     print(arg3)
>>> func(say='world') # Using keyword argument
>>> func(1, 'world')     # Using positional arguments
>>> func()               # Using default arguments
>>>
```

FUNCTION: ARBITRARY ARGUMENT

- Though seldomly used, but handy when defining a function without knowing the possible argument upfront
- Example:

```
>>> def myfunc(name='test', *args, **kwargs):
…     print(name)
…     print(args)
…     print(kwargs)
…
>>> myfunc()
>>> myfunc('test2', 1, 2, 3)
>>> myfunc('test3', 1, 2, arg3='hello', arg4='world')
```

FUNCTION: ANONYMOUS

- Anonymous function can be created using the keyword *lambda*
- Example:

```
>>> is_even = lambda n: False if n == 0 or n % 2 else True
…
>>> is_even(0)
>>> is_even(1)
>>> is_even(2)
>>>
```

- Typically used when you need a function object

OBJECT ORIENTED PROGRAMMING (OOP)

OOP: INTRODUCTION

- Programming paradigm based on the concept of data structure revolving around objects rather than actions
- Object contain attributes (variable within the scope of the object) and methods (functions within the object)
- The definition of object data structure is by using classes
- In Python, everything is an object
- Example:
  >>> a = 1
  …
  >>> type(a)

OOP: DEFINING A CLASS

- Using *class* keyword
- Example:

```
>>> class MyClass:
…     def __init__(self, say='what'):
…         self.say = say
…     def shout(self):
…         print(self.say)
…
>>> a = MyClass('Hello, world!')
>>> a.shout()
>>> a.say = 'What say you?'
>>> a.shout()
```

OOP: ARBITRARY DATA STRUCTURE

- Python allow you to create a new data structure
- Example:

```
>>> class Point:
...     def __init__(self, x, y):
...         self.x, self.y = x, y
...     def __add__(self, pt):
...         return Point(self.x+pt.x, self.y+pt.y)
...     def __sub__(self, pt):
...         return Point(self.x-pt.x, self.y-pt.y)
...     def __eq__(self, pt):
...         return self.x == pt.x and self.y == pt.y
```

OOP: INHERITANCE

- Derive a class from a base class
- Example:

```
>>> class Tyre:
…     def __init__(self, n=4):
…         self.numtyre = num
…
>>> class Color:
…     def __init__(self, c='red'):
…         self.color = c
…
```

- Example (cont):

```
>>> class Car(Tyre):
…     def __init__(self):
…         super(Car, self).__init__()
…
>>> a = Car()
>>> a.numtyre
```

- Show multiple inheritance quirks

ERRORS & EXCEPTIONS

ERRORS & EXCEPTIONS

- Two type of error messages
  - Syntax error
  - Exceptions
- Syntax error example:
  >>> if a ok
- Exceptions examples:
  >>> z
  >>> 4 + “1”
- Exception happens when the syntax is correct during execution but error is detected

EXCEPTION HANDLING

- Unlike syntax error, exception can be handle by your code
- Example:

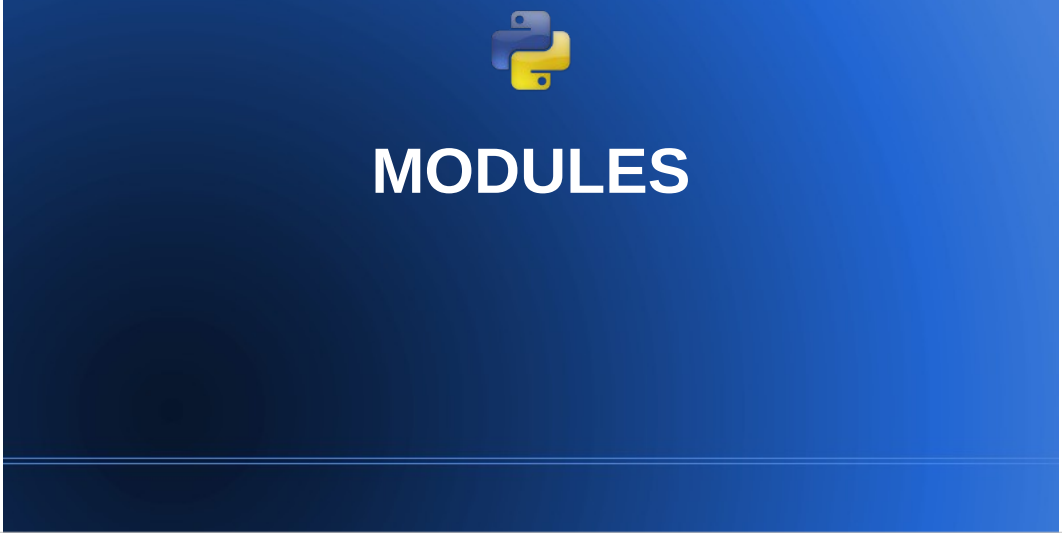
```
>>> try:
…     z / 0
… except NameError:
…     print(“Don't know what z is!!”)
… except ZeroDivisionError:
…     print(“Can't divide z with 0!”)
… except:
…     print(“Some other error”)
…
>>>
```

EXCEPTION HANDLING (CONT)

- Another example:
  ```
  >>> try:
  …     z / 0
  … except (NameError, ZeroDivisionError) as e:
  …     print(“Error: {0}.format(e.args[0]))
  … except:
  …     print('some other error')
  …
  ```
- There is also try-except-else and try-except-else-finally statement
- The *else* block will be evaluated after successful run of *try* block while the *finally* block will always executed

RAISING EXCEPTION

- Exception can be manually raised using the keyword *raise*
- Example:
  >>> raise NameError('duh! Var is not defined')
- Another example using custom error:
  >>> class MyCustomError(Exception):
  … pass
  …
  >>> raise MyCustomError('wow! my own error')


MODULES

MODULE: INTRODUCTION

- A module is a file (with .py extension) containing Python definitions and statements.
- The file name will become the module name and can be imported from your main program
- Technically, your main program is also a (special) module called “__main__” module.
- Python searches for modules in the current directory follows by the path defined in PYTHONPATH then in installation dependent default.
- Python comes with a lot of standard modules

MODULE: EXAMPLE

- Create a file called mymod.py (this will be your module)

```
def func1():
    print ('func1 from mymod')

def func2():
    print('func2 from mymod')
```

- Open Python shell from the same directory

```
>>> import mymod
>>> mymod.func1()
>>> mymod.func2()
```

- Module is accessed using dot notation

MODULE: PACKAGE

- Package is a way to structuring or organising modules into a more logical collection as you see fit

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

BATTERIES INCLUDED

BATTERIES INCLUDED

- Standard installation of Python comes with lots of built-in functions and an array of standard libraries.
- In some cases, these functions and libraries are all you need to build your application.
- These functions and libraries in most cases are well documented.
- While there are a lot to cover, we'll only walkthrough a subset.
- The focus for this section will be the ones you'll (probably) often use.

BUILT-IN FUNCTIONS

- enumerate([seq, start=0])
- raw_input([prompt])
- len(s)
- range([start,] stop[, step]), xrange()
- open(name[, mode])
- set([iterable])
- int(), oct(), hex(), bin()

STANDARD LIBRARIES

- sys
- os
- time
- datetime
- math
- decimal
- logging
- doctest & unittest