

PYTHON LANGUAGE

Web sites

- **A Byte of Python**
 - <https://python.swaroopch.com/>
- **Python 3 Tutorial**
 - http://www.python-course.eu/python3_course.php
- **The Python Tutorial**
 - <https://docs.python.org/3.5/tutorial/index.html>

Outline

3

- **Basics**
- **Operators, Expressions, and Control Flow**
- **Functions**
- **Data Structures**
- **Modules**
- **Object Oriented Programming**
- **Input Output**
- **Exceptions**
- **Standard Library**
- **More**

Basics: comments, literals

4

- **Comments**
 - Comments are any text to the right of the # symbol

```
print('Hello World') # Note that print is a function
or:
# Note that print is a function
print('Hello World')
```
- **Literal Constants**
 - A literal constant is a number like 5, 1.23, or a string like 'This is a string' or "It's a string!".
 - Quoting String with Single Quotes or double quotes
 - 字串用單引號，或雙引號都可以

Basics: numbers

5

□ Numbers

- Numbers are mainly of two types - integers and floats.
- An examples of an integer is 2 which is just a whole number. Examples of floating point numbers (or floats for short) are 3.23 and 52.3E-4.
 - The E notation indicates powers of 10. In this case, 52.3E-4 means $52.3 * 10^{-4}$.
- There is no separate long type. The int type can be an integer of any size.

print function

6

- `print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)`
 - Print *objects* to the text stream *file*, separated by *sep* and followed by *end*.
 - All non-keyword arguments are converted to strings like `str()` does and written to the stream, separated by *sep* and followed by *end*.
 - Both *sep* and *end* must be strings; they can also be `None`, which means to use the default values.

```
>>> x = 'spam'
>>> y = 99
>>> z = ['eggs']
>>> print(x, y, z) # Print three objects per defaults
spam 99 ['eggs']
>>> print(x, y, z, sep='') # Suppress separator
spam99['eggs']
>>> print(x, y, z, sep=', ') # Custom separator
spam, 99, ['eggs']
```

input function

7

□ Input from Keyboard

- If the input function is called, the program will be stopped until the user has given an input and has ended the input with the **return key**.
- input returns user input as a **string**.

```
while True:
    value = input("Integer, please [q to quit]: ")
    if value == 'q': # quit
        break
    number = int(value)
    if number % 2 == 0: # an even number
        continue
    print(number, "squared is", number*number)
```

```
Integer, please [q to quit]: 2
Integer, please [q to quit]: 3
3 squared is 9
Integer, please [q to quit]: q
```

Basics: strings

8

□ Strings

- Quoting String with Single Quotes or double quotes
 - 字串用單引號'，或雙引號"都可以
- Triple Quotes
 - specify **multi-line strings** using triple quotes - (""" or ''').
 - You can use single quotes and double quotes freely within the triple quotes.

```
'''This is a multi-line string. This is the first line.
This is the second line.
"What's your name?," I asked.
He said "Bond, James Bond."
'''
```

Basics: strings

9

- **Strings Are Immutable (不可變的)**
 - Once you have created a string, you cannot change it.
- **The format method**
 - <https://docs.python.org/3/library/string.html#formatstrings>
 - Sometimes we may want to construct strings from other information. This is where the `format()` method is useful.

```
age = 20
name = 'Mike'
print('{0} was {1} years old when he wrote this
book'.format(name, age))
print('Why is {0} playing with that python?'.format(name))
```

- Output:

```
Mike was 20 years old when he wrote this book
Why is Mike playing with that python?
```

Basics: strings

10

- `{}`: the numbers are optional

```
age = 20
name = 'Mike'
print('{} was {} years old when he wrote this book'.format(name, age))
print('Why is {} playing with that python?'.format(name))
```

which will give the same exact output as the previous program.

- `print()` always ends with an invisible “new line” character (`\n`)
 - To prevent this newline character from being printed, you can override the `end` parameter to `print`:

```
print("a", end="")
print("b", end="")
```

Output

```
ab
```

Basics: strings

11

□ Escape Sequences

- Suppose, you want to have a string which contains a single quote (')
- For example, the string is What's your name?.
- This can be done with the help of what is called an **escape sequence**.
- You specify the single quote as \' – notice the **backslash**.
- you can specify the string as 'What\'s your name?'
- Another way of specifying this specific string would be "What's your name?" i.e. using double quotes.

Basics: strings

12

- What if you wanted to specify a two-line string?
 - use a **triple quoted** string
 - or a newline character - \n
 - "This is the first line\nThis is the second line."
- A single backslash at the end of the line indicates that the string is continued in the next line, but no newline is added.

```
"This is the first sentence. \  
This is the second sentence."
```

is equivalent to

```
"This is the first sentence. This is the second sentence."
```

Basics: strings

13

□ Raw String

- If you need to specify some strings where no special processing such as escape sequences are handled, then what you need is to specify a raw string by prefixing `r` or `R` to the string.

```
r"Newlines are indicated by \n"
```

Strings as a sequence

14

□ The Index

- Because the elements of a string are a sequence, we can associate each element with an **index**, a location in the sequence:
- positive values count up from the left, beginning with **index 0**
- negative values count down from the right, starting with **-1**

characters	H	e	l	l	o		W	o	r	l	d
index	0	1	2	3	4	5	6	7	8	9	10
									...	-2	-1

FIGURE 4.1 The index values for the string `'Hello World'`.

Accessing an element

15

- A particular element of the string is accessed by the index of the element surrounded by square brackets []

```
hello_str = 'Hello World'
```

```
print(hello_str[1])    => prints e
```

```
print(hello_str[-1])  => prints d
```

```
print(hello_str[11])  => ERROR
```

Slicing

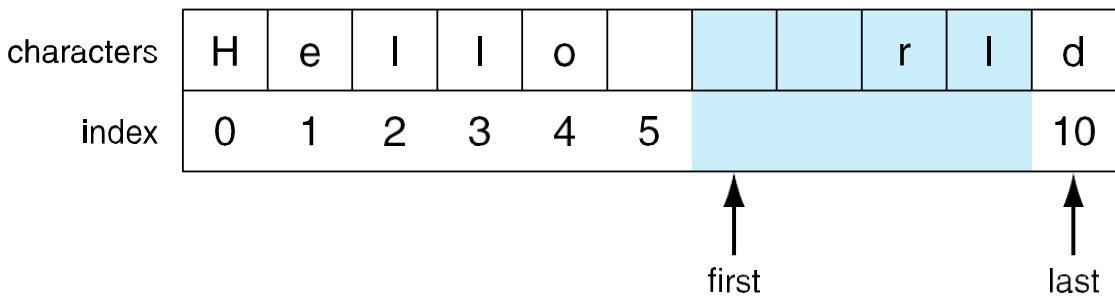
16

- slicing is the ability to select a subsequence of the overall sequence
- uses the syntax `[start : finish]`, where:
 - ▣ `start` is the index of where we start the subsequence
 - ▣ `finish` is the index of **one after** where we end the subsequence
- if either `start` or `finish` are not provided, it defaults to the beginning of the sequence for `start` and the end of the sequence for `finish`

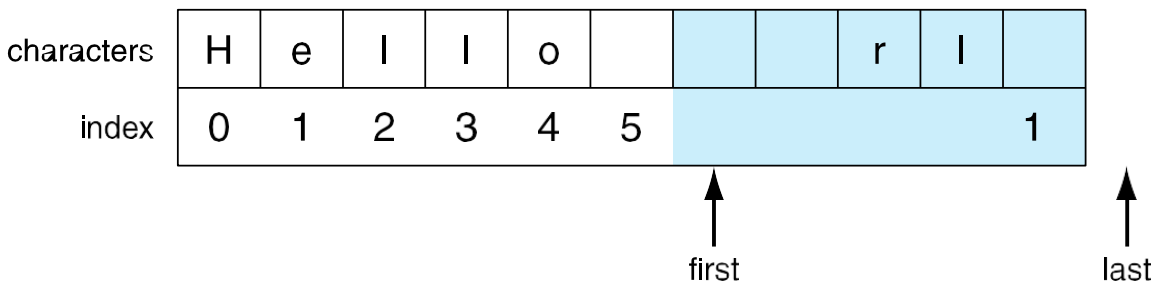
half open range for slices

- slicing uses what is called a half-open range
- the first index is included in the sequence
- the last index is one **after** what is included

```
helloString[6:10]
```



```
helloString[6:]
```



```
helloString[:5]
```

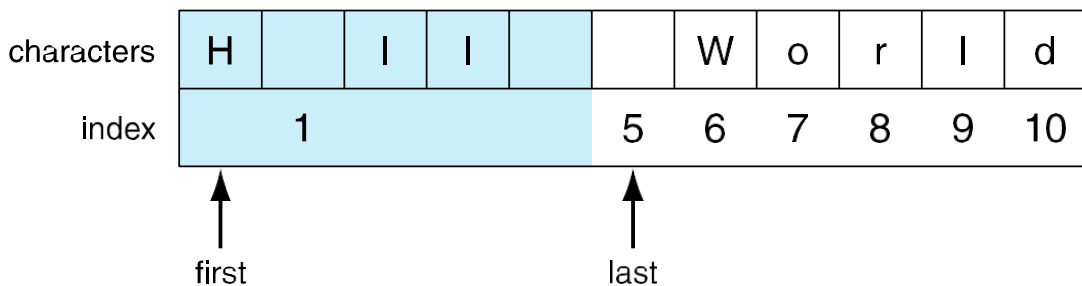


FIGURE 4.3 Two default slice examples.

```
helloString[-1]
```

Characters	H	e	l	l	o		W	o	r	l	d
Index	0	1	2	3	4	5	6	7	8	9	10
	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

↑
Last

```
helloString[3:-2]
```

Characters	H	e	l	l	o		W	o	r	l	d
Index	0	1	2	3	4	5	6	7	8	9	10

↑
First

↑
Last

Exercise

20

- Assume the variable `date` has been set to a string value of the form `mm/dd/yyyy`, for example `09/08/2010`. (Actual numbers would appear in the string.)
- Write a program to assign to a variable named `dayStr` the characters in `date` that contain the day. Then set a variable `day` to the integer value corresponding to the two digits in `dayStr`. Finally, output `day`.
 - ▣ That is, input “09/08/2010”, output 08

Extended Slicing

21

- also takes three arguments:
 - `[start:finish:step]`
 - defaults are:
 - `start` is beginning, `finish` is end, `step` is 1
- ```
my_str = 'hello world'
my_str[0:11:2] ⇒ 'hlowrd'
```
- every other letter

# Slicing with a step

22

```
In [6]: hello_str = "Hello World"
```

```
In [7]: hello_str[::2]
```

```
Out[7]: 'HloWrD'
```

```
In [8]: hello_str[::3]
```

```
Out[8]: 'HlWl'
```

```
In [9]: hello_str[::-1]
```

```
Out[9]: 'dlroW olleH'
```

```
In [10]: hello_str[::-2]
```

```
Out[10]: 'drWolH'
```

```
helloString[::2]
```

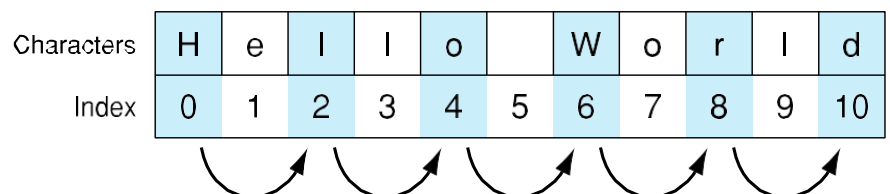


FIGURE 4.6 Slicing with a step.

# Exercise

23

- Given the strings `s1` and `s2`, not necessarily of the same length, create a new string consisting of alternating characters of `s1` and `s2`
  - ▣ that is, the first character of `s1` followed by the first character of `s2`, followed by the second character of `s1`, followed by the second character of `s2`, and so on.
- Once the end of either string is reached, the remainder of the longer string is added to the end of the new string.
- For example, if `s1` contained "abc" and `s2` contained "uvwxyz", then the new string should contain "aubvcwxyz". Associate the new string with the variable `s3`.

# Identifier Naming

24

- Variables are examples of identifiers.
  - ▣ The **first character** of the identifier must be a letter of the **alphabet** (uppercase ASCII or lowercase ASCII or Unicode character) or an underscore ('\_').
  - ▣ The rest of the identifier name can consist of letters (uppercase ASCII or lowercase ASCII or Unicode character), underscores ('\_') or digits (0-9).
  - ▣ Identifier names are **case-sensitive**.

# Data Types

25

- The basic types are numbers and strings
- Create our own types using classes
- **Object**
  - Python refers to anything used in a program as an *object*
  - Instead of saying ‘the something’, we say ‘the object’.
  - Python is strongly object-oriented in the sense that everything is an object including numbers, strings and functions.
- **None**
  - **None** is a special type in Python that represents nothingness.
  - For example, it is used to indicate that a variable has no value if it has a value of None.

26

OPERATORS, EXPRESSIONS  
AND CONTROL FLOW

# Operators

27

- **+** (plus)
  - $3 + 5$  gives 8. 'a' + 'b' gives 'ab'
- **-** (minus)
  - -5.2 gives a negative number and  $50 - 24$  gives 26
- **\*** (multiply)
  - $2 * 3$  gives 6. 'la' \* 3 gives 'lalala'.
- **\*\*** (power)
  - $3 ** 4$  gives 81 (i.e.  $3 * 3 * 3 * 3$ )
- **/** (divide)
  - $13 / 3$  gives 4.333333333333333
- **//** (floor division)
  - $13 // 3$  gives 4.
- **%** (modulo)
  - $13 \% 3$  gives 1.  $-25.5 \% 2.25$  gives 1.5.

# Operators

28

- **<<** (left shift)
  - $2 << 2$  gives 8 (i.e.  $10 \rightarrow 1000$ )
- **>>** (right shift)
  - $11 >> 1$  gives 5.
- **&, |, ^, ~** (bit-wise AND, OR, XOR, invert)
- **<, >, <=, >=, ==, !=**
- **not, and, or** (Boolean NOT, AND, OR)

## Operator precedence

| Operator                                                              | Description                                                             |
|-----------------------------------------------------------------------|-------------------------------------------------------------------------|
| lambda                                                                | Lambda expression                                                       |
| if – else                                                             | Conditional expression                                                  |
| or                                                                    | Boolean OR                                                              |
| and                                                                   | Boolean AND                                                             |
| not x                                                                 | Boolean NOT                                                             |
| in, not in, is, is not, <, <=, >, >=, !=, ==                          | Comparisons, including membership tests and identity tests              |
|                                                                       | Bitwise OR                                                              |
| ^                                                                     | Bitwise XOR                                                             |
| &                                                                     | Bitwise AND                                                             |
| <<, >>                                                                | Shifts                                                                  |
| +, -                                                                  | Addition and subtraction                                                |
| *, @, /, //, %                                                        | Multiplication, matrix multiplication division, remainder [5]           |
| +x, -x, ~x                                                            | Positive, negative, bitwise NOT                                         |
| **                                                                    | Exponentiation [6]                                                      |
| await x                                                               | Await expression                                                        |
| x[index], x[index:index], x(arguments...), x.attribute                | Subscription, slicing, call, attribute reference                        |
| (expressions...), [expressions...], {key: value...}, {expressions...} | Binding or tuple display, list display, dictionary display, set display |

## Control Flow

30

input returns user input as a string.

### □ The if statement

```
number = 23
guess = int(input('Enter an integer : '))
if guess == number:
 print('Congratulations, you guessed it.') # New block starts here
 print('(but you do not win any prizes!)') # New block ends here
elif guess < number:
 print('No, it is a little higher than that') # Another block
 # You can do whatever you want in a block ...
else:
 print('No, it is a little lower than that')
 # you must have guessed > number to reach here
print('Done')
```

- **elif** and **else** parts are optional.
- **colon :** is to declare the start of an indented block.
  - **if, elif, else, while, for, def, class** should be followed by an indented block.

# Control Flow

31

## □ The while Statement

- A while statement can have an optional **else** clause.

```
number = 23
running = True
while running:
 guess = int(input('Enter an integer : '))
 if guess == number:
 print('Congratulations, you guessed it.')
 running = False # this causes the while loop to stop
 elif guess < number:
 print('No, it is a little higher than that.')
 else:
 print('No, it is a little lower than that.')
else:
 print('The while loop is over.')
 # Do anything else you want to do here
print('Done')
```

True and False are boolean constants

# Control Flow

32

## □ The for loop

```
Python code
result = 0
for i in range(100):
 result += i
```

```
/* C code */
int result = 0;
for(int i=0; i<100;
i++){
 result += i;
}
```

- `range(100)` → `[0, 1, 2, ..., 99]`
- `range(start, stop, step)`
  - `range(1, 5, 2)` gives `[1, 3]`. (step =2)
- If you omit start, the range begins at 0. The only required value is stop.
  - `range(5) = range(0, 5)` gives the sequence `[0, 1, 2, 3, 4]`

```
In [4]: print(list(range(5)))
[0, 1, 2, 3, 4]
```



# Control Flow

33

## □ The for loop

- The **for .. in** statement is another looping statement which *iterates* over a sequence of objects i.e. go through each item in a sequence.

```
for i in range(1, 5):
 print(i)
else:
 print('The for loop is over')
```

```
Output:
1
2
3
4
The for loop is over
```

- `range(1, 5)` gives the sequence `[1, 2, 3, 4]`.

# Control Flow

34

## □ The for loop

- Note that `range()` generates a sequence of numbers, but it will generate **only one number at a time**, when the for loop requests for the next item.
  - If you want to see the full sequence of numbers immediately, use `list(range())`.
- `for i in range(1,5)` is equivalent to  
`for i in [1, 2, 3, 4]`
- The **else** part is optional.

# Control Flow

35

## □ The break Statement

- The break statement is used to *break* out of a loop statement

```
while True:
 s = input('Enter something : ')
 if s == 'quit':
 break
 print('Length of the string is', len(s))
print('Done')
```

```
Output:
Enter something : When the work is done
Length of the string is 21
Enter something : use Python!
Length of the string is 12
Enter something : quit
Done
```

# Control Flow

36

## □ The continue Statement

- The continue statement is used to tell Python to skip the rest of the statements in the current loop block and to continue to the next iteration of the loop.

```
while True:
 s = input('Enter something : ')
 if s == 'quit':
 break
 if len(s) < 3:
 print('Too small')
 continue
 print('Input is of sufficient length')
```

```
Output:
Enter something : a
Too small
Enter something : 12
Too small
Enter something : abc
Input is of sufficient length
Enter something : quit
```

# Files

37

- You can open and use files for reading or writing by creating an object of the **file** class
- When you are finished with the file, you call the **close** method to tell Python that we are done using the file.

- File modes

| Mode | How Opened            | File Exists                                                          | File Does Not Exist                 |
|------|-----------------------|----------------------------------------------------------------------|-------------------------------------|
| 'r'  | <b>read-only</b>      | <b>Opens that file</b>                                               | <b>Error</b>                        |
| 'w'  | write-only            | Clears the file contents                                             | Creates and opens a new file        |
| 'a'  | <b>write-only</b>     | <b>File contents left intact and new data appended at file's end</b> | <b>Creates and opens a new file</b> |
| 'r+' | read and write        | Reads and overwrites from the file's beginning                       | Error                               |
| 'w+' | <b>read and write</b> | <b>Clears the file contents</b>                                      | <b>Creates and opens a new file</b> |
| 'a+' | read and write        | File contents left intact and read and write at file's end           | Creates and opens a new file        |

Example (save as using\_file.py):

```
poem = '''\
Programming is fun
When the work is done
if you wanna make your work also fun:
 use Python!
'''
f = open('poem.txt', 'w') # open for writing
f.write(poem) # write text to file
f.close() # close the file
f = open('poem.txt')
if no mode is specified, read mode is assumed by default

while True:
 line = f.readline()
 if len(line) == 0: # Zero length indicates EOF
 break
 print(line, end='')
f.close() # close the file
```

38

```
Output:
$ python3 using_file.py
Programming is fun
When the work is done
if you wanna make your work also fun:
 use Python!
```

# Example: Reverse file lines

39

```
input_file = open("input.txt", "r")
output_file = open("output.txt", "w")

for line_str in input_file:
 new_str = ''
 line_str = line_str.strip() # get rid of carriage return
 for char in line_str:
 new_str = char + new_str # concat at the left (reverse)
 print(new_str, file=output_file) # print to output_file

 # include a print to shell so we can observe progress
 print('Line: {:12s} reversed is: {:s}'.format(line_str, new_str))
input_file.close()
output_file.close()
```

<https://docs.python.org/3/library/stdtypes.html#string-methods>

## Python string methods

40

|                                    |                                      |
|------------------------------------|--------------------------------------|
| capitalize( )                      | rstrip( [chars] )                    |
| center( width[, fillchar] )        | partition( sep )                     |
| count( _ )                         | replace( old, new[, count_] )        |
| decode( [encoding[, errors]] )     | rfind( sub[, start[, end]] )         |
| encode( _ )                        | ri ( [, start[, end]] )              |
| endswith( suffix[, start[, end]] ) | rjust( width[, fillchar] )           |
| expandtabs( _tabsize )             | rpartition( sep )                    |
| find( sub[, start[, end]] )        | rsplit( [sep[, maxsplit]] )          |
| i ( _ )                            | rs ( _chars )                        |
| isalnum( )                         | split( [sep[, maxsplit]] )           |
| is ( )                             | splitlines( [keepends_] )            |
| isdigit( )                         | startswith( prefix[, start[, end]] ) |
| isl ( )                            | s ( _chars )                         |
| isspace( )                         | swapcase( )                          |
| isil ( )                           | til ( )                              |
| isupper( )                         | translate( table[, deletechars] )    |
| jo ( )                             | uppe ( )                             |
| lower( )                           | zfill( width )                       |
| lj ( )                             |                                      |

TABLE 4.2 Python String Methods

## String strip() method

41

- `str.strip([chars]);` (剥掉特定字元)
- **Parameters**
  - **chars** – The characters to be removed from beginning or end of the string.
  - The method **strip()** returns a copy of the string in which all chars have been stripped from the beginning and the end of the string (**default whitespace characters**).
- **Example**
  - `str = "oooooooothis is string example....wow!!!oooooooo"`
  - `print str.strip('o')`
  - **Output:** this is string example....wow!!!

## String split() method

42

- `str.split(sep=None, maxsplit=-1)`
  - Return a list of the words in the string, using `sep` as the delimiter string.
  - If `maxsplit` is given, at most `maxsplit` splits are done (thus, the list will have at most `maxsplit+1` elements).
  - If `maxsplit` is not specified or `-1`, then there is no limit on the number of splits (all possible splits are made).
  - by default, if no argument is provided, split is on any whitespace character (tab, blank, etc.)

```
>>> '1,2,3'.split(',')
['1', '2', '3']
>>> '1,2,3'.split(',', maxsplit=1)
['1', '2,3']
>>> '1,2,,3,.'.split(',')
['1', '2', '', '3', '']
```

```
>>> '1 2 3'.split()
['1', '2', '3']
>>> '1 2 3'.split(maxsplit=1)
['1', '2 3']
>>> ' 1 2 3 '.split()
['1', '2', '3']
```

# Files: **with** keyword

43

- It is good practice to use the **with** keyword when dealing with file objects.
- This has the advantage that the file is properly closed after its block finishes, even if an exception is raised on the way.

```
>>> with open('workfile', 'r') as f:
... read_data = f.read()
>>> f.closed
True
```

44

FUNCTIONS

# Functions

45

- Functions are defined using the **def** keyword.

```
def say_hello():
 print('Hello World!')
End of function #

say_hello() # call the function
say_hello() # call the function again
```

## Naming

module\_name, package\_name, ClassName, method\_name, ExceptionName, **function\_name**, GLOBAL\_CONSTANT\_NAME, **global\_var\_name**, instance\_var\_name, **function\_parameter\_name**, **local\_var\_name**.

```
Output:
Hello World!
Hello World!
```

# Function Parameters

46

- *Parameters*: the names given in the function definition
- *Arguments*: the values you supply in the function call

```
def print_max(a, b):
 if a > b:
 print(a, 'is maximum')
 elif a == b:
 print(a, 'is equal to', b)
 else:
 print(b, 'is maximum')

print_max(3, 4) # directly give literal values
x = 5
y = 7
print_max(x, y) # give variables as arguments
```

```
Output:
4 is maximum
7 is maximum
```

# Local Variables

47

- When you declare variables inside a function definition, they are not related in any way to other variables with the same names used outside the function
  - i.e. variable names are *local* to the function.

```
x = 50
def func(x):
 print('x is', x)
 x = 2
 print('Changed local x to', x)

func(x)
print('x is still', x)
```

```
Output:
x is 50
Changed local x to 2
x is still 50
```

# Using The global Statement

48

- If you want to assign a value to a name defined at the **top level of the program** (i.e. not inside any kind of scope such as functions or classes), then you have to tell Python that the name is not local, but it is *global*.

```
x = 50
def func():
 global x
 print('x is', x)
 x = 2
 print('Changed global x to', x)

func()
print('Value of x is', x)
```

```
Output:
x is 50
Changed local x to 2
Value of x is 2
```



# Default Argument Values

49

```
def say(message, times = 1):
 print(message * times)

say('Hello')
say('World', 5)
```

```
Output:
Hello
WorldWorldWorldWorldWorld
```

# Keyword Arguments

50

- You can give values for such parameters by naming them
  - ▣ this is called *keyword arguments* - we use the name (keyword) instead of the position
  - ▣ we do not need to worry about the order of the arguments.

```
def func(a, b=5, c=10):
 print('a is', a, 'and b is', b, 'and c is', c)

func(3, 7)
func(25, c=24)
func(c=50, a=100)
```

```
Output:
a is 3 and b is 7 and c is 10
a is 25 and b is 5 and c is 24
a is 100 and b is 5 and c is 50
```

# VarArgs parameters

51

- Take any number of parameters, this can be achieved by using the stars

```
def total(initial=5, *numbers, **keywords):
 count = initial
 print(numbers)
 print(keywords)
 for number in numbers:
 count += number
 for key in keywords:
 count += keywords[key]
 return count
```

```
Output:
(1, 2, 3)
{'vegetables': 50, 'fruits': 100}
166
```

```
print(total(10, 1, 2, 3, vegetables=50, fruits=100))
```

- A starred parameter (\*param): all the positional arguments from that point till the end are collected as a **tuple** called 'param'.
- A double-starred parameter (\*\*param): all the keyword arguments from that point till the end are collected as a **dictionary** called 'param'.

# The lambda() Function

52

- In Python, a *lambda function* is an anonymous function expressed as a single statement.
- Let's first make an example that uses normal functions. We'll define the function `edit_story()` with arguments
  - words—a list of words
  - func—a function to apply to each word in words

```
>>> def edit_story(words, func):
... for word in words:
... print(func(word))

>>> stairs = ['thud', 'meow', 'thud', 'hiss']
>>> def enliven(word):
... return word.capitalize() + '!'

>>> edit_story(stairs, enliven)
```

```
Out:
Thud!
Meow!
Thud!
Hiss!
```

# The lambda() Function

53

- The `enliven()` function was so brief that we could replace it with a lambda:

```
>>> edit_story(stairs, lambda word: word.capitalize() + '!')
Thud!
Meow!
Thud!
Hiss!
```

- The lambda takes one argument, which we call `word` here.
- Everything between the colon and the terminating parenthesis is the definition of the function.

# Tuples

54

- A tuple is a sequence of immutable Python objects.
- Tuples are sequences, just like lists.
- The differences between tuples and lists are, the tuples cannot be changed unlike lists.
- Tuples use parentheses `()`, whereas lists use square brackets `[]`.

```
tup1 = ('physics', 'chemistry', 1997, 2000);
tup2 = (1, 2, 3, 4, 5);
tup3 = "a", "b", "c", "d";
```

- The empty tuple is written as two parentheses containing nothing

```
tup1 = ();
```

- To write a tuple containing a single value you have to include a comma, even though there is only one value –

```
tup1 = (50,);
```

# Keyword-only Parameters

55

- Specify certain keyword parameters to be available as keyword-only and **not** as positional arguments, declared **after a starred parameter**

```
def total(initial=5, *numbers, extra_number):
 count = initial
 for number in numbers:
 count += number
 count += extra_number
 print(count)
```

```
total(10, 1, 2, 3, extra_number=50)
total(10, 1, 2, 3)
Raises error because we have not supplied a
default argument value for 'extra_number'
```

Output:

66

Traceback (most recent call last):

File "keyword\_only.py", line 12, in <module>

total(10, 1, 2, 3)

TypeError: total() needs keyword-only argument extra\_number

# return Statement

56

- The return statement is used to *return* from a function i.e. break out of the function.
- We can optionally *return a value* from the function as well.

```
def maximum(x, y):
 if x > y:
 return x
 elif x == y:
 return 'The numbers are equal'
 else:
 return y

print(maximum(2, 3))
```

Output:

3

# DocStrings

57

- DocStrings are an important tool that you should make use of since it helps to document the program better and makes it easier to understand.

```
def printMax(x, y):
 '''Prints the maximum of two numbers.

 The two values must be integers.'''
 x = int(x) # convert to integers, if possible
 y = int(y)
 if x > y:
 print(x, 'is maximum')
 else:
 print(y, 'is maximum')

printMax(3, 5)
print(printMax.__doc__)
```

```
Output:
5 is maximum
Prints the maximum of two numbers.
```

```
The two values must be integers.
```

58

DATA STRUCTURES

# Classes and objects

59

- Python is strongly **object-oriented** in the sense that everything is an object including numbers, strings and functions.
- A **class** creates a new *type*. (class用來定義資料型態)
- **Objects** are *instances* of the class. (object類似變數)
- **Fields:** Variables that belong to an object or class.
  - Fields are of two types
    - **instance variables:** belong to each instance/object of the class
    - **class variables:** belong to the class itself
- **Methods:** functions defined for use with respect to that class/object only.
- **Attributes:** the fields and methods of that class.

# List

60

- Holds an ordered collection of items
  - you can store a *sequence* of items in a list.
  - put commas (,) in between them.
  - The list of items should be enclosed in square brackets []
  - Once you have created a list, you can add, remove or search for items in the list.
    - Since we can add and remove items, we say that a list is a *mutable* data type i.e. [this type can be altered](#).

# Similarities with strings

61

- Concatenate: + (but only of lists)
- Repeat: \*
- indexing (the [ ] operator)
- slicing ([:])
- membership (the in operator)
- len (the length operator)
  - Create an empty list: a = []
  - len(a) → 0
- **check if a list is empty?**

```
if not a:
 print("List is empty")
```

## Example (save as using\_list.py)

62

```
This is my shopping list #
shoplist = ['apple', 'mango', 'carrot', 'banana']
print('I have', len(shoplist), 'items to purchase.')
```

```
print('These items are:', end=' ')
for item in shoplist:
 print(item, end=' ')
```

```
print('\nI also have to buy rice.')
```

```
shoplist.append('rice')
print('My shopping list is now', shoplist)
```

```
print('I will sort my list now')
```

```
shoplist.sort()
print('Sorted shopping list is', shoplist)
```

```
print('The first item I will buy is', shoplist[0])
olditem = shoplist[0]
del shoplist[0]
print('I bought the', olditem)
print('My shopping list is now', shoplist)
```

append() and sort()  
are methods of list

Output:

```
$ python3 using_list.py
I have 4 items to purchase.
These items are: apple mango carrot banana
I also have to buy rice.
My shopping list is now ['apple', 'mango', 'carrot', 'banana', 'rice']
I will sort my list now
Sorted shopping list is ['apple', 'banana', 'carrot', 'mango', 'rice']
The first item I will buy is apple
I bought the apple
My shopping list is now ['banana', 'carrot', 'mango', 'rice']
```

# Operators

63

```
[1, 2, 3] + [4] ⇒ [1, 2, 3, 4]
```

```
[1, 2, 3] * 2 ⇒ [1, 2, 3, 1, 2, 3]
```

```
1 in [1, 2, 3] ⇒ True
```

```
1 not in [1, 2, 3] ⇒ False
```

```
[1, 2, 3] < [1, 2, 4] ⇒ True
```

compare index to index, first difference determines the result

# Iteration

64

- You can iterate through the elements of a list like you did with a string:

```
>>> my_list = [1,3,4,8]
>>> for element in my_list: # iterate through list elements
 print(element ,end=' ') # prints on one line

1 3 4 8
>>>
```



# List Comprehension

65

- List comprehensions are used to derive a new list from an existing list.
  - For example, derive a new list by specifying the manipulation to be done ( $2*i$ ) when some condition is satisfied ( $i > 2$ ).

Example (save as `list_comprehension.py`):

```
listone = [2, 3, 4]
listtwo = [2*i for i in listone if i > 2]
print(listtwo)
```

Output:

```
$ python3 list_comprehension.py
[6, 8]
```

# List Comprehension

66

- Example
  - Use the `json` module and its `loads` function invoked on each line in the sample file

```
import json
path = 'ch02/usagov_bitly_data2012-03-16-1331923249.txt'
records = [json.loads(line) for line in open(path)]
```

```
In [18]: records[0]
Out[18]:
{'u'a': u'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/535.11 (KHTML, like
Gecko) Chrome/17.0.963.78 Safari/535.11',
 u'al': u'en-US,en;q=0.8',
 u'c': u'US',
 u'cy': u'Danvers',
 u'g': u'A6qOVH',
 u'gr': u'MA',
 ...
```

# enumerate

67

```
names = ['Alice', 'Bob', 'Cindy']
index = 0
while index < len(names):
 print('{} {}'.format(index, names[index]))
 index += 1
```

Without  
enumerate

```
names = ['Alice', 'Bob', 'Cindy']
for index, element in enumerate(names):
 print('{} {}'.format(index, element))
```

With  
enumerate

Output:  
1 Alice  
2 Bob  
3 Cindy

## Enumerate and zip

- **enumerate**- Iterate over indices and items of a list

```
alist = ['a1', 'a2', 'a3']
for i, a in enumerate(alist):
 print (i, a)
```

```
1 a1
2 a2
3 a3
```

- **zip**- Iterate over two lists in parallel

```
alist = ['a1', 'a2', 'a3']
blist = ['b1', 'b2', 'b3']
for a, b in zip(alist, blist):
 print (a, b)
```

```
a1 b1
a2 b2
a3 b3
```

# Empty list

69

- Create an empty list
  - ▣ `lst = []` # faster
  - ▣ `lst = list()` # slower
- Check if a list is empty
  - ▣ Empty lists evaluate to False in boolean contexts

```
if not myList:
 print "Nothing here"
```

or

```
if len(my_list) == 0:
 print "my_list is empty"
```

## exercise

70

- Associate True with the variable `has_dups` if the list `list1` has any duplicate elements (that is if any element appears more than once), and False otherwise.

## exercise

71

- Write a function named **remove\_dup** that has one parameter, a list whose elements are of type int. The function returns the list without duplicates.
  - Hint:
    - Create a empty list, no\_dup
    - For item in lst
    - if item not in no\_dup
    - no\_dup.append(item)

## Tuple

72

- As similar to lists, but without the extensive functionality that the list class gives you.
  - They are **faster** in processing as compared to lists.
- One major feature of tuples is that they are **immutable** like strings i.e. you cannot modify tuples.
- Tuples are defined by specifying items separated by commas within an **optional** pair of parentheses.
  - I prefer always having them to make it obvious that it is a tuple.
  - For example, print(1,2,3) and print((1,2,3)) mean two different things.
- **Tuple with 0 item**
  - Using an empty pair of parentheses such as myempty = ().
- **Tuple with 1 item**
  - Using a comma following the first (and only) item, ex. singleton = (2,)

Example (save as using\_tuple.py):

```
zoo = ('python', 'elephant', 'penguin')
remember the parentheses are optional

print('Number of animals in the zoo is', len(zoo))
new_zoo = 'monkey', 'camel', zoo
print('Number of cages in the new zoo is', len(new_zoo))
print('All animals in new zoo are', new_zoo)
print('Animals brought from old zoo are', new_zoo[2])
print('Last animal brought from old zoo is', new_zoo[2][2])
print('Number of animals in the new zoo is', len(new_zoo)-1+len(new_zoo[2]))
```

Output:

```
$ python3 using_tuple.py
Number of animals in the zoo is 3
Number of cages in the new zoo is 3
All animals in new zoo are ('monkey', 'camel', ('python', 'elephant',
'penguin'))
Animals brought from old zoo are ('python', 'elephant', 'penguin')
Last animal brought from old zoo is penguin
Number of animals in the new zoo is 5
```

## Passing tuples around

74

- Return two different values from a function?
  - use a tuple
  - The usage of `a, b = <some expression>` interprets the result of the expression as a tuple with two values.

```
>>> def get_error_details():
... return (2, 'second error details')
...
>>> errnum, errstr = get_error_details()
>>> errnum
2
>>> errstr
'second error details'
```

# Passing tuples around

75

- If you want to interpret the results as `(a, <everything else>)`,
- then you just need to [star it](#) just like you would in function parameters

```
>>> a, *b = [1, 2, 3, 4]
>>> a
1
>>> b
[2, 3, 4]
```

- This also means the fastest way to swap two variables in Python is:

```
>>> a = 5; b = 8
>>> a, b = b, a
>>> a, b
(8, 5)
```

# Dictionary

76

- Associate **keys** (name) with **values** (details).
  - `d = {key1 : value1, key2 : value2 }`
  - The key-value pairs are separated by a **colon**
  - The pairs are separated themselves by **commas** and all this is enclosed in a pair of curly braces.
- Note that the **key** must be **unique**.
- You can use only immutable objects (like strings) for the keys of a dictionary but you can use either immutable or mutable objects for the values of the dictionary.
- Remember that key-value pairs are not ordered
- **Keyword Arguments and Dictionaries**
  - On a different note, if you have used keyword arguments (`**param`) in your functions, you have already used dictionaries!

| Operation                                             | Interpretation                                               |
|-------------------------------------------------------|--------------------------------------------------------------|
| <code>D = {}</code>                                   | Empty dictionary                                             |
| <code>D = {'name': 'Bob', 'age': 40}</code>           | Two-item dictionary                                          |
| <code>E = {'cto': {'name': 'Bob', 'age': 40}}</code>  | Nesting                                                      |
| <code>D = dict(name='Bob', age=40)</code>             | Alternative construction techniques:                         |
| <code>D = dict([('name', 'Bob'), ('age', 40)])</code> | keywords, key/value pairs, zipped key/value pairs, key lists |
| <code>D = dict(zip(keyslst, valueslist))</code>       |                                                              |
| <code>D = dict.fromkeys(['name', 'age'])</code>       |                                                              |
| <code>D['name']</code>                                | Indexing by key                                              |
| <code>E['cto']['age']</code>                          |                                                              |
| <code>'age' in D</code>                               | Membership: key present test                                 |
| <code>D.keys()</code>                                 | Methods: all keys,                                           |
| <code>D.values()</code>                               | all values,                                                  |
| <code>D.items()</code>                                | all key+value tuples,                                        |
| <code>D.copy()</code>                                 | copy (top-level),                                            |


|                                              |                                                |
|----------------------------------------------|------------------------------------------------|
| <code>D.clear()</code>                       | clear (remove all items),                      |
| <code>D.update(D2)</code>                    | merge by keys,                                 |
| <code>D.get(key, default?)</code>            | fetch by key, if absent default (or None),     |
| <code>D.pop(key, default?)</code>            | remove by key, if absent default (or error)    |
| <code>D.setdefault(key, default?)</code>     | fetch by key, if absent set default (or None), |
| <code>D.popitem()</code>                     | remove/return any (key, value) pair; etc.      |
| <code>len(D)</code>                          | Length: number of stored entries               |
| <code>D[key] = 42</code>                     | Adding/changing keys                           |
| <code>del D[key]</code>                      | Deleting entries by key                        |
| <code>list(D.keys())</code>                  | Dictionary views (Python 3.X)                  |
| <code>D1.keys() &amp; D2.keys()</code>       |                                                |
| <code>D.viewkeys(), D.viewvalues()</code>    | Dictionary views (Python 2.7)                  |
| <code>D = {x: x*2 for x in range(10)}</code> | Dictionary comprehensions (Python 3.X, 2.7)    |

# Basic Dictionary Operations

79

```
% python
Make a dictionary
>>> D = {'spam': 2, 'ham': 1, 'eggs': 3}

Fetch a value by key
>>> D['spam']
2
Order is "scrambled"
>>> D
{'eggs': 3, 'spam': 2, 'ham': 1}
>>> len(D) # Number of entries in dictionary
3
>>> 'ham' in D # Key membership test alternative
True
>>> list(D.keys()) # Create a new list of D's keys
['eggs', 'spam', 'ham']
```



The left-to-right order of keys in a dictionary will almost always be different from what you originally typed.

# Changing Dictionaries in Place

80

```
>>> D
{'eggs': 3, 'spam': 2, 'ham': 1}

>>> D['ham'] = ['grill', 'bake', 'fry'] # Change entry
(value=list)
>>> D
{'eggs': 3, 'spam': 2, 'ham': ['grill', 'bake', 'fry']}

>>> del D['eggs'] # Delete entry
>>> D
{'spam': 2, 'ham': ['grill', 'bake', 'fry']}

>>> D['brunch'] = 'Bacon' # Add new entry
>>> D
{'brunch': 'Bacon', 'spam': 2, 'ham': ['grill', 'bake', 'fry']}
```



# Dictionary Example

81

```
ab = { 'Swaroop' : 'swaroop@swaroopch.com',
 'Larry' : 'larry@wall.org',
 'Matsumoto' : 'matz@ruby-lang.org',
 'Spammer' : 'spammer@hotmail.com'
 }
print("Swaroop's address is", ab['Swaroop'])
Deleting a key-value pair
del ab['Spammer']

print('\nThere are {0} contacts in the address-book\n'.format(len(ab)))

for name, address in ab.items():
 print('Contact {0} at {1}'.format(name, address))

Adding a key-value pair
ab['Guido'] = 'guido@python.org'
if 'Guido' in ab:
 print("\nGuido's address is", ab['Guido'])
```

## Output:

```
Swaroop's address is swaroop@swaroopch.com
There are 3 contacts in the address-book
Contact Swaroop at swaroop@swaroopch.com
Contact Matsumoto at matz@ruby-lang.org
Contact Larry at larry@wall.org

Guido's address is guido@python.org
```

# Building dictionaries faster

82

- `zip` creates pairs from two parallel lists
  - `zip("abc", [1, 2, 3])` yields `[('a', 1), ('b', 2), ('c', 3)]`

```
alist = ['a1', 'a2', 'a3']
blist = ['b1', 'b2', 'b3']
for a, b in zip(alist, blist):
 print(a, b)
```



```
a1 b1
a2 b2
a3 b3
```

- That's good for building dictionaries. We call the `dict` function which takes a list of pairs to make a dictionary
  - `dict(zip("abc", [1, 2, 3]))` yields
  - `{'a': 1, 'c': 3, 'b': 2}`

# dict comprehension

83

```
>>> a_dict = {k:v for k,v in enumerate('abcdefg')}
>>> a_dict
{0: 'a', 1: 'b', 2: 'c', 3: 'd', 4: 'e', 5: 'f', 6: 'g'}
>>> b_dict = {v:k for k,v in a_dict.items()} # reverse key-value pairs
>>> b_dict

{'a': 0, 'c': 2, 'b': 1, 'e': 4, 'd': 3, 'g': 6, 'f': 5}
>>> sorted(b_dict) # only sorts keys
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> b_list = [(v,k) for v,k in b_dict.items()] # create list
>>> sorted(b_list) # then sort
[('a', 0), ('b', 1), ('c', 2), ('d', 3), ('e', 4), ('f', 5), ('g', 6)]
```

# Receiving Tuples and Dictionaries in Functions

84

- There is a special way of receiving parameters to a function as a **tuple** or a **dictionary** using the **\*** or **\*\*** prefix respectively.

```
def total(initial=5, *numbers, **keywords):
 count = initial
 print(numbers)
 print(keywords)
 for number in numbers:
 count += number
 for key in keywords:
 count += keywords[key]
 return count
```

```
print(total(10, 1, 2, 3, vegetables=50, fruits=100))
```

```
Output:
(1, 2, 3)
{'vegetables': 50, 'fruits': 100}
166
```

- Because we have a **\*** prefix on the **numbers** variable, all extra arguments passed to the function are stored in **numbers** as a **tuple**.
- A double-starred parameter (**\*\*keywords**): all the keyword arguments from that point till the end are collected as a **dictionary** called 'keywords'.

# Sequence

85

- Lists, tuples and strings are examples of sequences
- The major features are **membership tests**, (i.e. the `in` and `not in` expressions) and **indexing operations**, which allow us to fetch a particular item in the sequence directly.
  - ▣ lists, tuples and strings, also have a **slicing** operation which allows us to retrieve a slice of the sequence i.e. a part of the sequence.

```
shoplist = ['apple', 'mango', 'carrot', 'banana']
name = 'swaroop'
Indexing or 'Subscription' operation
print('Item 0 is', shoplist[0])
Out[4]: Item 0 is apple
print('Item 1 is', shoplist[1])
Out[5]: Item 1 is mango
print('Item 2 is', shoplist[2])
Out[6]: Item 2 is carrot
print('Item 3 is', shoplist[3])
Out[7]: Item 3 is banana
print('Item -1 is', shoplist[-1])
Out[8]: Item -1 is banana
print('Item -2 is', shoplist[-2])
Out[9]: Item -2 is carrot
print('Character 0 is', name[0])
Out[10]: Character 0 is s
```

```
shoplist = ['apple', 'mango',
'carrot', 'banana']
```

```
Slicing on a list
```

```
print('Item 1 to 3 is', shoplist[1:3])
```

```
Out[11]: Item 1 to 3 is ['mango', 'carrot']
```

```
print('Item 2 to end is', shoplist[2:])
```

```
Out[12]: Item 2 to end is ['carrot', 'banana']
```

```
print('Item 1 to -1 is', shoplist[1:-1])
```

```
Out[13]: Item 1 to -1 is ['mango', 'carrot']
```

```
print('Item start to end is', shoplist[:])
```

```
Out[14]: Item start to end is ['apple', 'mango', 'carrot', 'banana']
```

```
print('First 3 items', shoplist[:3])
```

```
Out[15]: First 3 items ['apple', 'mango', 'carrot']
```

```
Slicing on a string
```

```
print('characters 1 to 3 is', name[1:3])
```

```
Out[16]: characters 1 to 3 is wa
```

```
print('characters 2 to end is', name[2:])
```

```
Out[17]: characters 2 to end is aroop
```

```
print('characters 1 to -1 is', name[1:-1])
```

```
Out[18]: characters 1 to -1 is waroo
```

```
print('characters start to end is', name[:])
```

```
Out[19]: characters start to end is swaroop
```

```
name = 'swaroop'
```

# Sequence

89

- Python starts counting numbers from 0.
  - Hence, `shoplist[0]` fetches the first item
  - The index can also be a negative number
    - `shoplist[-1]` refers to the **last item** in the sequence
    - `shoplist[-2]` fetches the **second last item** in the sequence.
- Slicing operation
  - `shoplist[1:3]` returns a slice of the sequence starting at position 1, includes position 2 but stops at position 3.
  - `shoplist[:]` returns a copy of the **whole sequence**.
  - Negative numbers are used for positions from the end of the sequence.
    - For example, `shoplist[:-1]` will return a slice of the sequence which **excludes the last item** of the sequence but contains everything else.
  - A third argument for the slice, which is the **step** for the slicing (by default, the step size is 1)
    - `shoplist[::2]`, get the items with position 0, 2, ...

# Set

90

- Sets are *unordered* collections of simple objects.
- In a set, no two elements are identical. That is, a set consists of elements each of which is **unique** compared to the other elements
- Using sets, you can test for membership, whether it is a subset of another set, find the intersection between two sets, and so on.

```
>>> bri = set(['brazil', 'russia', 'india'])
>>> 'india' in bri
True
>>> 'usa' in bri
False
>>> bric = bri.copy()
>>> bric.add('china')
>>> bric.issuperset(bri)
True
>>> bri.remove('russia')
>>> bri & bric # OR bri.intersection(bric)
{'brazil', 'india'}
```

# Creating a set

91

- Set can be created in one of two ways:
  - constructor: `set(iterable)` where the argument is iterable

```
my_set = set('abc')
my_set → {'a', 'b', 'c'}
```
  - shortcut: `{}`, braces where the elements have no colons (to distinguish them from dicts)

```
my_set = {'a', 'b', 'c'}
```
- A set can consist of a mixture of different types of elements

```
my_set = {'a', 1, 3.14159, True}
```

# no duplicates

92

- Duplicates are automatically removed

```
my_set = set("aabbccdd")
print(my_set)
→ {'a', 'c', 'b', 'd'}
```

## example

```
>>> null_set = set() # set() creates the empty set
>>> null_set
set()
>>> a_set = {1,2,3,4} # no colons means set
>>> a_set
{1, 2, 3, 4}
>>> b_set = {1,1,2,2,2} # duplicates are ignored
>>> b_set
{1, 2}
>>> c_set = {'a', 1, 2.5, (5,6)} # different types is OK
>>> c_set
{(5, 6), 1, 2.5, 'a'}
>>> a_set = set("abcd") # set constructed from iterable
>>> a_set
{'a', 'c', 'b', 'd'} # order not maintained!
```

## exercise

94

- Write a function named **remove\_dup** that has one parameter, a list whose elements are of type int. The function returns the list without duplicates.
  - Hint: convert the list to a set, and then convert to list

## common set operators

95

- Most data structures respond to these:
  - ▣ `len(my_set)`
    - the number of elements in a set
  - ▣ `element in my_set`
    - boolean indicating whether element is in the set
  - ▣ `for element in my_set:`
    - iterate through the elements in `my_set`

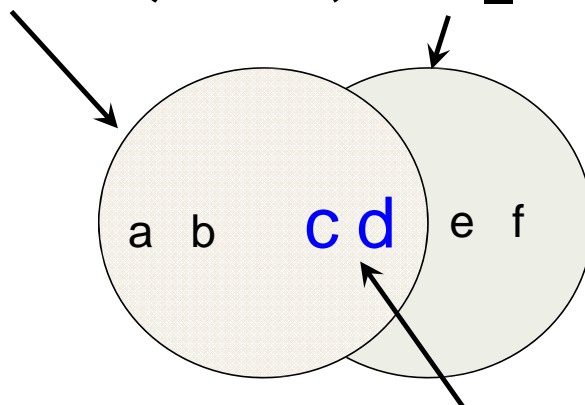
- set comprehension

```
>>> a_set = {ch for ch in 'to be or not to be'}
>>> a_set
{' ', 'b', 'e', 'o', 'n', 'r', 't'} # set of unique characters
>>> sorted(a_set)
[' ', 'b', 'e', 'n', 'o', 'r', 't']
```

## method: intersection, op: &

96

`a_set=set("abcd")`      `b_set=set("cdef")`



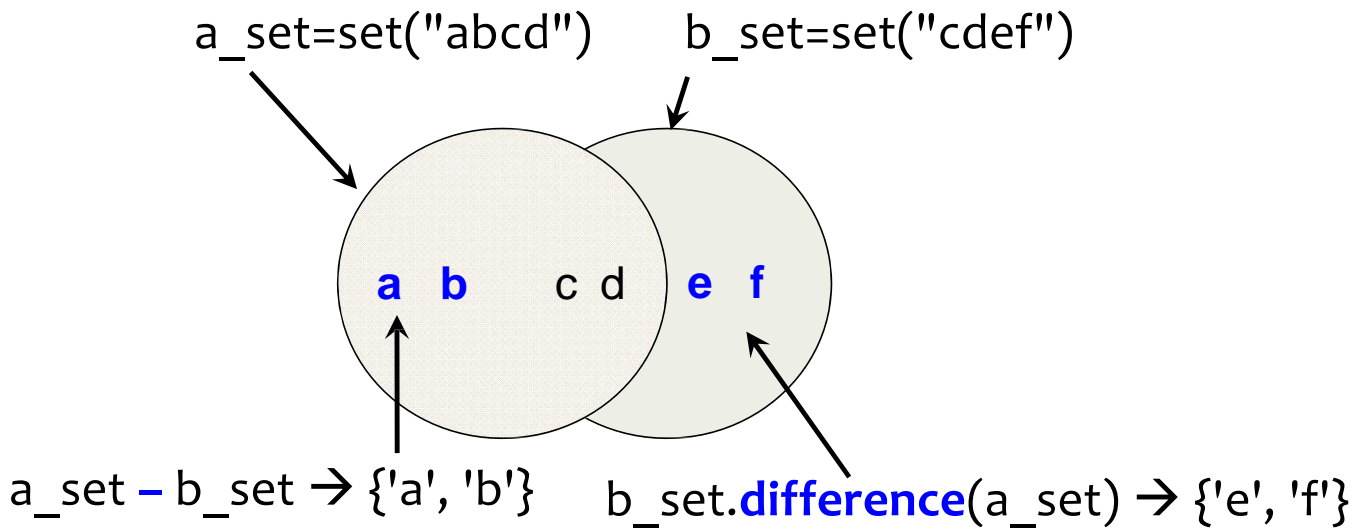
`a_set & b_set` → {'c', 'd'}

`b_set.intersection(a_set)` → {'c', 'd'}



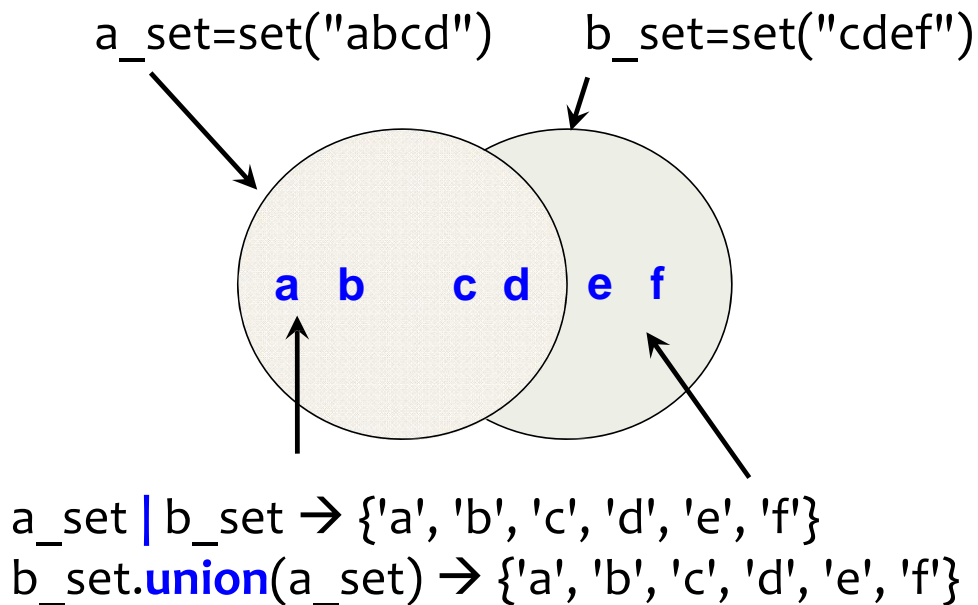
## method: difference op: -

97



## method: union, op: |

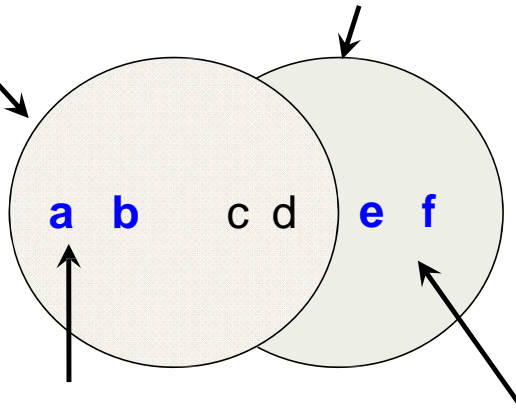
98



method: symmetric\_difference, op: ^

99

a\_set=set("abcd")      b\_set=set("cdef")



a\_set ^ b\_set → {'a', 'b', 'e', 'f'}

b\_set.symmetric\_difference(a\_set) → {'a', 'b', 'e', 'f'}

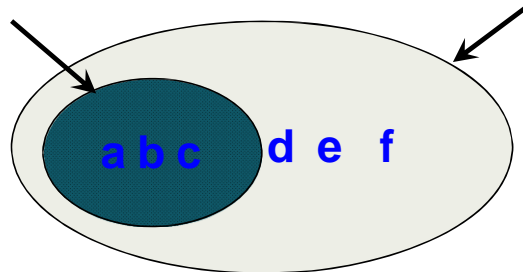
method: issubset, op: <=

method: issuperset, op: >=

100

small\_set=set("abc")

big\_set=set("abcdef")



small\_set <= big\_set → True

big\_set >= small\_set → True

## Other Set Ops

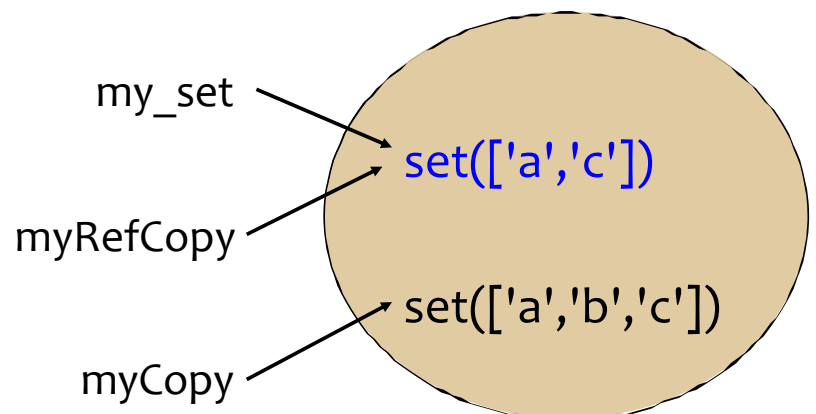
101

- `my_set.add("g")`
  - ▣ adds to the set, no effect if item is in set already
- `my_set.clear()`
  - ▣ emptys the set
- `my_set.remove("g")` versus `my_set.discard("g")`
  - ▣ `remove` throws an error if "g" isn't there. `discard` doesn't care. Both remove "g" from the set
- `my_set.copy()`
  - ▣ returns a shallow copy of `my_set`

## Copy vs. assignment

102

```
my_set=set {'a', 'b', 'c'}
my_copy=my_set.copy()
my_ref_copy=my_set
my_set.remove('b')
```



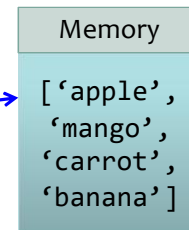
# References

103

- When you create an object and assign it to a variable, the variable only *refers* to the object and does not represent the object itself!
  - ▣ That is, the variable name points to that part of your computer's memory where the object is stored.
  - ▣ This is called **binding** the name to the object.

```
shoplist = ['apple', 'mango',
 'carrot', 'banana']
```

shoplist



# References

104

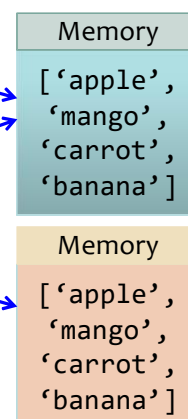
- Remember that if you want to make a copy of a list or such kinds of sequences or complex objects (not simple objects such as integers)
  - ▣ then you have to use the **slicing operation** to make a copy.
  - ▣ If you just assign the variable name to another name, both of them will “refer” to the same object and this could be trouble if you are not careful.

```
shoplist = ['apple', 'mango',
 'carrot', 'banana']
listA = shoplist
listB = shoplist[0:]
or shoplist[:] or copy() method
```

shoplist

listA

listB



# Example

105

```
print('Simple Assignment')
shoplist = ['apple', 'mango', 'carrot', 'banana']
mylist = shoplist # mylist is just another name pointing to the same object!

del shoplist[0] # I purchased the first item, so I remove it from the list
print('shoplist is', shoplist)
print('mylist is', mylist)
notice that both shoplist and mylist both print the same list without
the 'apple' confirming that they point to the same object

print('Copy by making a full slice')
mylist = shoplist[:] # make a copy by doing a full slice
del mylist[0] # remove first item

print('shoplist is', shoplist)
print('mylist is', mylist)
notice that now the two lists are different
```

## Output:

```
Simple Assignment
shoplist is ['mango', 'carrot', 'banana']
mylist is ['mango', 'carrot', 'banana']
Copy by making a full slice
shoplist is ['mango', 'carrot', 'banana']
mylist is ['carrot', 'banana']
```

# Difference between `==` and `is`

106

- **is** will return True if two variables **point to the same object**
- **==** if two objects have the same value.

```
In [1]: a = [1, 2, 3]
In [2]: b = a
In [3]: b == a
Out[3]: True
```

```
In [4]: b is a
Out[4]: True
```

```
In [5]: b = a[:]
In [6]: b == a
Out[6]: True
```

```
In [7]: b is a
Out[7]: False
```

```
In [8]: 1000 is 10**3
Out[8]: False
```

```
In [9]: 1000 == 10**3
Out[9]: True
```

# More About Strings

107

- The strings that you use in program are all objects of the class str.
  - There are many useful methods of this class, see help(str)

Example:

```
name = 'Swaroop' # This is a string object
if name.startswith('Swa'):
 print('Yes, the string starts with "Swa"')
if 'a' in name:
 print('Yes, it contains the string "a"')
if name.find('war') != -1:
 print('Yes, it contains the string "war"')
delimiter = '_'
mylist = ['Brazil', 'Russia', 'India', 'China']
print(delimiter.join(mylist))
```

**Output:**

```
Yes, the string starts with "Swa"
Yes, it contains the string "a"
Yes, it contains the string "war"
Brazil_*_Russia_*_India_*_China
```

108

## MODULES

# Modules

109

- Module
  - The highest-level program organization unit
    - which packages program code and data for reuse, and provides self-contained namespaces that minimize variable name clashes across your programs.
  - Modules typically correspond to Python program files.
    - Each file (.py) is a module, and modules import other modules to use the names they define.
  - May contain a number of functions.
  - Modules might also correspond to extensions coded in external languages such as C, Java, or C#, and even to directories in package imports.

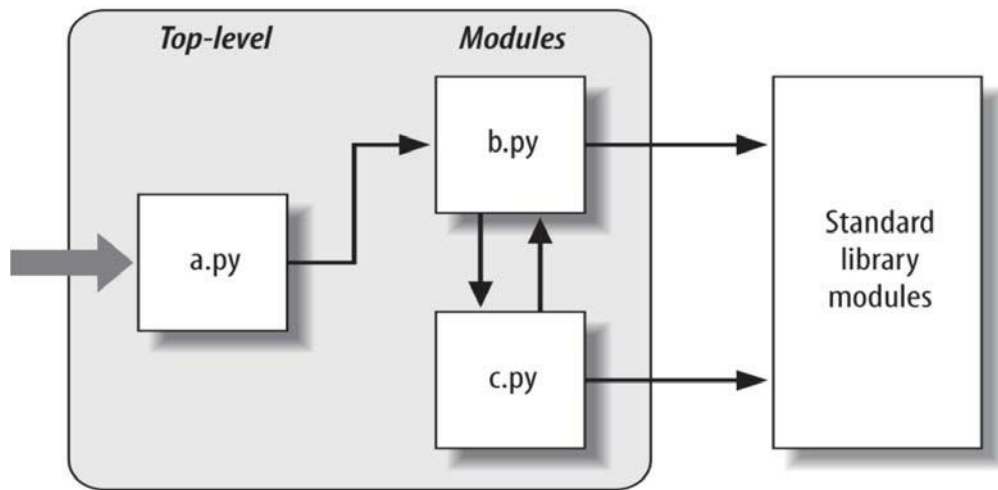
# Python Program Architecture

110

- At a base level, a Python program consists of text files, with **one main top-level file**, and zero or more supplemental files known as **modules**.
  - The top-level (a.k.a. script) file contains the main flow of control of your program—this is the file you run to launch your application.
  - The module files are libraries of tools used to collect components used by the top-level file, and possibly elsewhere.
  - Top-level files use tools defined in module files, and modules use tools defined in other modules.
  - A file **imports** a module to gain access to the tools it defines, which are known as its **attributes—variable names attached to objects such as functions**. (object.attribute notation)

# Python Program Architecture

111



## Import statement

112

- The file *a.py* is chosen to be the top-level file

```
import b # File a.py
b.spam('gumby') # Prints "gumby spam"
```

```
def spam(text): # File b.py
 print(text, 'spam')
```

- A Python import statement, gives the file *a.py* access to everything defined by top-level code in the file *b.py*.
- The code *import b* roughly means:
  - ▣ Load the file *b.py* (unless it's already loaded), and give me access to all its attributes (functions) through the name *b*.
  - ▣ The **module name** also becomes a **variable** (object) assigned to the loaded module.



# Import statement

113

## □ import X

- Imports the module X, and creates a reference to that module in the current namespace.
- Then you need to define [completed module path to access a particular attribute or method](#) from inside the module (e.g.: X.name or X.attribute)

# Import statement example

114

## □ Generate pseudo-random numbers

- **Source code:** Lib/random.py
- This module implements pseudo-random number generators for various distributions.
- <https://docs.python.org/3.5/library/random.html>
- random.seed(a=None, version=2)
  - Initialize the random number generator.
  - If a is omitted or None, the current system time is used.
- random.randint(a, b)
  - Return a random integer N such that  $a \leq N \leq b$ .

```
import random
random.seed()
a = random.randint(1, 1000)
print(a)
```

# How Imports Work

115

- Import is different from C #include.
- Perform three distinct steps
  - Find the module's file.
  - Compile it to byte code (if needed)
  - Run the module's code to build the objects it defines.
- The Module Search Path
  1. The home directory of the program
  2. PYTHONPATH directories (if set)
  3. Standard library directories
  4. The contents of any .pth files (if present)
  5. The site-packages home of third-party extensions
  - Ultimately, the concatenation of these four components becomes `sys.path`,

# How Imports Work

116

```
Example (save as using_sys.py):
import sys
print('The command line arguments are:')
for i in sys.argv:
 print(i)
print('\nThe PYTHONPATH is', sys.path, '\n')
```

```
$ python3 using_sys.py we are arguments
The command line arguments are:
using_sys.py
we
are
arguments

The PYTHONPATH is ['/home/cclee/tmp', '/usr/lib/python3.5.zip',
'/usr/lib/python3.5', '/usr/lib/python3.5/plat-x86_64-linux-gnu',
'/usr/lib/python3.5/lib-dynload', '/usr/local/lib/python3.5/dist-
packages', '/usr/lib/python3/dist-packages']
```

# Byte-compiled .pyc files

117

- Importing a module is a relatively costly affair, so Python does some tricks to make it faster.
- One way is to create *byte-compiled* files with the extension `.pyc`.
  - These byte-compiled files are platform-independent.
  - These `.pyc` files are usually created in the same directory as the corresponding `.py` files.

# from ... import ...

118

- Imports names (**attributes**) from a module directly into the importing module's symbol table.
- In general, **you should avoid using this statement** and use the `import` statement instead since your program will avoid name clashes and will be more readable.
  - 儘量少用 `from...import ...` 寫法 · 以避免名稱衝突
  - To avoid the conflict, you can use **`import...as`**

```
from math import sqrt
print("Square root of 16 is", sqrt(16))
```

## import ... as ...

119

- 為匯入的模組取別名，可以使用**import as**

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Give an alias

- This means that when you see `np.arange`, this is a reference to the `arange` function in NumPy.

## from X import \*

120

- Imports the module `X`, and creates references to all public objects defined by that module in the current namespace.
  - That is, this imports all names except those beginning with an underscore (`_`). (private)
  - In other words, after you've run this statement, you can simply use a plain name to refer to things defined in module `X`.
    - But `X` itself is not defined, so `X.name` doesn't work.
  - And if name was already defined, it is replaced by the new version. And if name in `X` is changed to point to some other object, your module won't notice.

# Import examples

121

- Package **skimage**: Image Processing SciKit (Toolbox for SciPy)
- Subpackage Module: **viewer**
  - Method: **ImageViewer()**

```
import skimage
from skimage import data
img = data.coffee()
viewer = skimage.viewer.ImageView(img)
viewer.show()
```

You should import a module, not package

x

Err:

AttributeError: module 'skimage' has no attribute 'viewer'

# Import examples

```
from skimage.viewer import ImageViewer
from skimage import data
img = data.coffee()
viewer = ImageViewer(img)
viewer.show()
```

✓

from [module]  
import [attribute]

```
from skimage import viewer
from skimage import data
img = data.coffee()
viewer = viewer.ImageView(img)
viewer.show()
```

✓

from [package]  
import [module]



# Import examples

123

```
import skimage.viewer as iv
from skimage import data
img = data.coffee()
viewer = iv.ImageViewer(img)
viewer.show()
```



```
import skimage.viewer.ImageViewer as iv
from skimage import data
img = data.coffee()
viewer = iv(img)
viewer.show()
```



Err:

```
ImportError: No module named 'skimage.viewer.ImageViewer'
```

# Module's name

124

- Every Python module has its `__name__` defined.
- If this is `'__main__'`, that implies that the module is being run standalone by the user.

Example (save as `using_name.py`):

```
if __name__ == '__main__':
 print('This program is being run by itself')
else:
 print('I am being imported from another module')
```

Output:

```
$ python3 using_name.py
This program is being run by itself
$ python3
>>> import using_name
I am being imported from another module
>>>
```

# Making Your Own Modules

125

- Every Python program is also a module.
  - The module should be placed either in the **same directory** as the program from which we import it, or in one of the **directories listed in `sys.path`**.

```
Example (save as mymodule.py):
def sayhi():
 print('Hi, this is mymodule speaking.')
__version__ = '0.1'
```

```
Another module (save as
mymodule_demo.py):
import mymodule
mymodule.sayhi()
print ('Version', mymodule.__version__)
```

```
Output:
$ python3 mymodule_demo.py
Hi, this is mymodule speaking.
Version 0.1
```

# Making Your Own Modules

126

- Here is a version utilizing the **from..import** syntax

```
from mymodule import sayhi, __version__
sayhi()
print('Version', __version__)
```

- The output is same as the output of `mymodule_demo.py`.
  - Notice that if there was already a `__version__` name declared in the module that imports `mymodule`, there would be a **clash**.
  - Hence, **it is always recommended to prefer the import statement** even though it might make your program a little longer.
  - You could also use:

```
from mymodule import *
```

    - This will import **all public names** such as `sayhi` but **would not import `__version__`** because it starts with **double underscores**.

private to  
its class

# The dir function

127

- Use dir function to list the identifiers that an object defines.
  - For example, for a module, the identifiers include the functions, classes and variables defined in that module.
  - When you supply a module name to the dir() function, it returns the list of the names defined in [that module](#).
  - When no argument is applied to it, it returns the list of names defined in the [current module](#).

```
cclee@Snoopy:~/tmp$ python3
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__',
 '__spec__']
>>> import sys
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__',
 '__spec__', 'sys']
>>>
```

# The dir function

128

```
>>> import sys
>>> dir(sys)
['_displayhook__', '__doc__', '__egginsert', '__excepthook__',
 '__name__', '__package__', '__plen', '__stderr__', '__stdin__',
 '__stdout__', '_clear_type_cache', '_current_frames', '_getframe',
 '_mercurial', 'api_version', 'argv', 'builtin_module_names',
 'byteorder', 'call_tracing', 'callstats', 'copyright', 'displayhook',
 'dont_write_bytecode', 'exc_clear', 'exc_info', 'exc_type',
 'excepthook', 'exec_prefix', 'executable', 'exit', 'flags',
 'float_info', 'float_repr_style', 'getcheckinterval',
 'getdefaultencoding', 'getdlopenflags', 'getfilesystemencoding',
 'getprofile', 'getrecursionlimit', 'getrefcount', 'getsizeof',
 'gettrace', 'hexversion', 'last_traceback', 'last_type', 'last_value',
 'long_info', 'maxint', 'maxsize', 'maxunicode', 'meta_path', 'modules',
 'path', 'path_hooks', 'path_importer_cache', 'platform', 'prefix',
 'ps1', 'ps2', 'py3kwarning', 'setcheckinterval', 'setdlopenflags',
 'setprofile', 'setrecursionlimit', 'settrace', 'stderr', 'stdin',
 'stdout', 'subversion', 'version', 'version_info', 'warnoptions']
>>>
```



```
>>> import __builtin
>>> dir(__builtin_)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BufferError',
'BytesWarning', 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception',
'False', 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError',
'ImportWarning', 'IndentationError', 'IndexError', 'KeyError', 'KeyboardInterrupt',
'LookupError', 'MemoryError', 'NameError', 'None', 'NotImplemented',
'NotImplementedError', 'OSError', 'OverflowError', 'PendingDeprecationWarning',
'ReferenceError', 'RuntimeError', 'RuntimeWarning', 'StandardError', 'StopIteration',
'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'True', 'TypeError',
'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError',
'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning',
'ZeroDivisionError', '_', '__debug__', '__doc__', '__import__', '__name__', '__package__',
'abs', 'all', 'any', 'apply', 'basestring', 'bin', 'bool', 'buffer', 'bytearray', 'bytes', 'callable', 'chr',
'classmethod', 'cmp', 'coerce', 'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir',
'divmod', 'enumerate', 'eval', 'execfile', 'exit', 'file', 'filter', 'float', 'format', 'frozenset',
'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'intern', 'isinstance',
'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'long', 'map', 'max', 'memoryview', 'min',
'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'raw_input',
'reduce', 'reload', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod',
'str', 'sum', 'super', 'tuple', 'type', 'unichr', 'unicode', 'vars', 'xrange', 'zip']
```

## Packages

130

- Hierarchy of organizing your programs
  - Variables usually go inside functions.
  - Functions and global variables usually go inside modules.
- Packages are just a convenience to hierarchically organize modules.
  - Packages are **just folders of modules** with a special **`__init__.py`** file that indicates to Python that this folder is special because it contains Python modules.
  - `__init__.py`
    - In the simplest case, `__init__.py` can just be an **empty** file,
    - but it can also **execute initialization code** for the package
    - or set the **`__all__`** variable

|              |                                        |
|--------------|----------------------------------------|
| 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   |                                        |
| ...          |                                        |

## Packages

- if a package's `__init__.py` code defines a list named `__all__`
  - it is taken to be the list of module names that should be imported when `from package import *` is encountered.
  - For example, the file `sound/effects/__init__.py` could contain the following code:
    - `__all__ = ["echo", "surround", "reverse"]`
    - This would mean that `from sound.effects import *` would import the three named submodules of the sound package.
- If `__all__` is not defined,
  - the statement `from sound.effects import *` does *not* import all submodules from the package `sound.effects` into the current namespace;
  - it only ensures that the package `sound.effects` has been imported.

# logging module

133

- Used to save some debugging or important messages

Save as use\_logging.py:

```
import os, platform, logging
if platform.platform().startswith('Windows'):
 logging_file = os.path.join(os.getenv('HOMEDRIVE'),
 os.getenv('HOMEPATH'), 'test.log')
else:
 logging_file = os.path.join(os.getenv('HOME'), 'test.log')

print("Logging to", logging_file)
logging.basicConfig(
 level=logging.DEBUG,
 format='%(asctime)s : %(levelname)s : %(message)s',
 filename = logging_file,
 filemode = 'w',
)
logging.debug("Start of the program")
logging.info("Doing something")
logging.warning("Dying now")
```

# logging module

134

Output:

```
$ python3 use_logging.py
Logging to C:\Users\swaroop\test.log
```

If we check the contents of test.log, it will look something like this:

```
2012-10-26 16:52:41,339 : DEBUG : Start of the program
2012-10-26 16:52:41,339 : INFO : Doing something
2012-10-26 16:52:41,339 : WARNING : Dying now
```

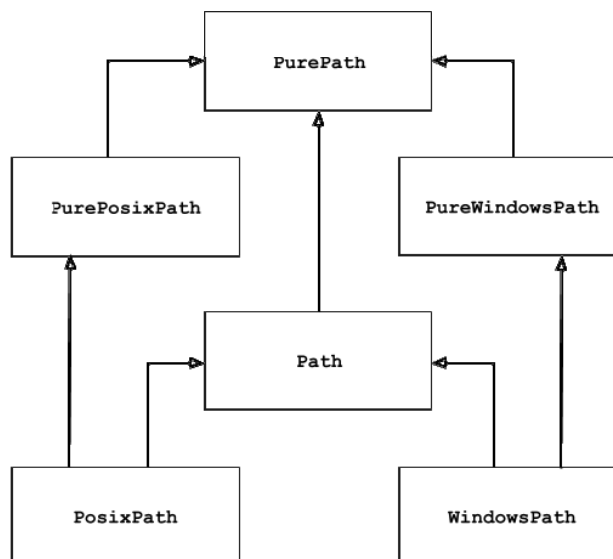
# pathlib module

135

- High-level path objects
  - <https://docs.python.org/3.5/library/pathlib.html>
  - This module offers classes representing filesystem paths with semantics appropriate for different operating systems.
  - Path classes are divided between **pure paths**, which provide purely computational operations without I/O, and **concrete paths**, which inherit from pure paths but also provide I/O operations.

# pathlib module

136



# os.path module

137

- Low-level path manipulations
  - <https://docs.python.org/3.5/library/os.path.html#module-os.path>
  - All of these functions accept either only bytes or only string objects as their parameters.
  - The result is an object of the same type, if a path or file name is returned.