Python Programming

Ch.01 Python Introduction

Instructor : 楊禎文

What is Python

- Invented in the Netherlands, early 90s by Guido van Rossum
- Named after Monty Python

What is Python

- Open source general-purpose language
- Considered a scripting language, but is much more
  - Great interactive environment
- Scalable, object oriented and functional
  - Easy to integrate with other languages
  - https://wiki.python.org/moin/IntegratingPythonWithOtherLanguages
- Used by Google from the beginning
- Increasingly popular

Why Python

- Designed to make it very readable
  - Being readable also makes it easier to learn and remember, hence more writeable
- Has a gentle learning curve that makes you productive sooner
  - Yet it has depths that you can explore as you gain expertise
- The program is much smaller than other static languages
- Python runs almost everywhere and has a lot of useful utilities in its standard library

Print message using C/C++

- Using C

```
#include <stdio.h>
int main(int argc, char *argv[]) {
    int language = 1;
    printf("Language %d: I am C! Behold me and tremble!\n", language);
    return 0;
}
```

- Using C++

```
#include <iostream>
using namespace std;
int main() {
    int language = 2;
    cout << "Language " << language << \
    ": I am C++! Pay no attention to that C behind the curtain!" << \
    endl;
    return(0);
}
```


Print message using Java/Python

- Using Java

```
public class Demo {
    public static void main (String[] args) {
        int language = 1;
        System.out.format("Language %d: I am Java! Scarier than C!\n", language);
    }
}
```

- Using Python

```
language = 2
print("Language %s: I am Python. What's for supper?" % language)
```

Who uses Python?

Python 2 vs. Python 3

- Python 2.0 was released on 16 October 2000
  - Has been around forever and is preinstalled on Linux and Apple computers
  - End Of Life date for last Python 2.7 was initially set at 2015, then postponed to 2020
  - Python 2 is the past
- Python 3.0 was released on 3 December 2008
  - Backwards-incompatible, bundle the hard fixes together
  - Many of its major features have been backported to the backwards-compatible Python 2.6.x and 2.7.x version series
  - Python 3 is the future

Install Python

- Linux distributions
  - Using repository (yum for RedHat, apt for Debina …)
  - apt-get install python3 idle3 (Debian)
- Windows
  - Download latest version from https://www.python.org/downloads/windows/
  - Using “Customize installation” to change install path
- Mac OS
  - Download latest version from https://www.python.org/downloads/mac-osx/
- Setting up PATH environment variable
  - Using shortcut(link) in Windows

Running Python

- The interactive interpreter that comes with Python gives you the capability to experiment with small programs
  - $ python3

    …

    >>> print(61)

    61

    >>>
- Store your Python programs in text files, normally with the .py extension, and run them by typing python followed by those filenames
  - $ python3 demo.py

Printing to the Screen

- The simplest way to produce output is using the print function where you can pass zero or more expressions separated by commas
- Print function converts the expressions you pass into a string and writes the result to standard output
  - print ("Python is a great language,", " try it!")

Reading Keyboard Input

- input() function read data from keyboard
- eval() function return the result of the evaluated expression
- >>> print(input('What is your name? '))
  What is your name? John
  John
  >>> print(eval(input('Do some math: ')))
  Do some math: 2+2*5
  12

Python IDLE

- Python's Integrated Development and Learning Environment
  - Coded in 100% pure Python, using the tkinter GUI toolkit
  - Cross-platform, works mostly the same on Windows, Unix, and Mac OS X
  - Python shell window with colorizing of code input, output, and error messages
  - Multi-window text editor with multiple undo, Python colorizing, smart indent, call tips, auto completion, and other features
  - Debugger with persistent breakpoints, stepping, and viewing of global and local namespaces

Using Anaconda

- Anaconda is the leading open data science platform powered by Python
- The open source version of Anaconda is a high performance distribution of Python and R and includes over 100 of the most popular Python, R and Scala packages for data science.
- Can access to over 720 packages that can easily be installed with conda package manager
- With Anaconda you can run multiple versions of Python in isolated environments
  - https://www.continuum.io/downloads

Install Anaconda

- Download installer and follow the instructions
  - https://www.continuum.io/downloads
- Anaconda Prompt
  - using conda utility
  - conda list : list installed packages
  - conda update --all
    - Update all installed packages
  - conda search pandas
    - Search packages using regex
- Using IPython

Spyder

- Scientific PYthon Development EnviRonment
- Powerful interactive development environment for the Python language
  - with advanced editing, interactive testing, debugging and introspection features
- Numerical computing environment
  - support IPython (enhanced interactive Python interpreter)
  - support popular Python libraries such as NumPy (linear algebra), SciPy (signal and image processing) or matplotlib (interactive 2D/3D plotting)

Project Jupyter

- Project Jupyter is an open source project
- Born out of the IPython Project in 2014
- Support interactive data science and scientific computing across all programming languages
- Jupyter will always be 100% open source software, free for all to use and released under the liberal terms of the modified BSD license
- Starting the Notebook Server
  - jupyter notebook
    - This will print some information about the notebook server in your terminal, including the URL of the web application (by default, http://localhost:8888)

Python References

- Python 3 documentation
  - https://docs.python.org/3/index.html
- The Python Tutorial
  - https://docs.python.org/3/tutorial/index.html
- Python Enhancement Proposals
  - https://www.python.org/dev/peps/
- Style Guide for Python Code
  - https://www.python.org/dev/peps/pep-0008/

Ch.02 Python Basic Syntax and Variable Types

Basic Syntax

First Python Program

- Interactive Mode
  - Invoke the Python interpreter
  - >>> print ("Hello, Python!")
    Hello, Python!
- Script Mode
  - Write a Python program in a script. Python files have the extension ".py"
  - Edit a new python file test.py
    - print("Hello, Python!")
  - Run python program
    - python3 test.py

Python Identifiers

- An identifier is a name used to identify a variable, function, class, module or other object
- An identifier starts with a letter A-Z or a-z or an underscore (_) followed by zero or more alphanumeric characters or underscores
- Python does not allow punctuation characters such as @, $, and % within identifiers
- Python is a case sensitive programming language. Thus, Hello and hello are two different identifiers

Identifier Naming Conventions

- Class names start with an uppercase letter
- All other identifiers start with a lowercase letter
- Starting an identifier with a single leading underscore indicates that the identifier is private
- Starting an identifier with two leading underscores indicates a strong private identifier
  - If the identifier also ends with two trailing underscores, the identifier is a language defined special name

Reserved Words

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

Lines and Indentation (1)

- Python provides no braces to indicate blocks of code for class and function definitions or flow control
- Blocks of code are denoted by line indentation, which is strictly enforced
- The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount
  - All the continuous lines indented with same number of spaces would form a block

Lines and Indentation (2)

- Correct syntax
  - ```
    if True:
        print("True")
    else:
        print("False")
    ```
- The following block generates an error –
  - ```
    if True:
        print("Answer")
        print("True")
    else:
        print("Answer")
      print("False")
    ```

Multi-Line Statements

- Statements typically end with a new line
- Use the line continuation character (\) to denote that the line should continue
  - total = item_one + \
    item_two + \
    item_three
- Statements contained within the [], {}, or () brackets do not need to use the line continuation character
  - days = ['Monday', 'Tuesday', 'Wednesday',
    'Thursday', 'Friday']

Quotation in Python

- Python accepts single ('), double (") and triple (''' or """) quotes to denote string literals, as long as the same type of quote starts and ends the string
- The triple quotes are used to span the string across multiple lines
- Examples
  - word = 'hello'
  - sentence = "This is a sentence"
  - paragraph = """This is a paragraph. It is made up of multiple lines and sentences."""

Comments

- A hash sign (#) that is not inside a string literal is the beginning of a comment. All characters after the #, up to the end of the physical line, are part of the comment and the Python interpreter ignores them
  - # First comment
    print ("Hello, Python!") # second comment
- Python does not have multiple-line commenting feature. You have to comment each line individually
  - # This is a comment
    # This is a comment, too
    # This is a comment, too

Group of Multiple Statements

- Groups of individual statements, which make a single code block are called suites in Python
- Compound or complex statements, such as if, while, def, and class require a header line and a suite
- Header lines begin the statement (with the keyword) and terminate with a colon (:) and are followed by one or more lines which make up the suite
- Example
  - if expression :
      suite
    elif expression :
      suite
    else :
      suite

Some Concepts

- Blank Lines
  - A line containing only whitespace, possibly with a comment, is known as a blank line and Python totally ignores it
- Multiple Statements on a Single Line
  - The semicolon (;) allows multiple statements on a single line given that no statement starts a new code block
  - import sys; x = 'foo'; sys.stdout.write(x + '\n')

Variable Types

Assigning Values to Variables

- Python variables do not need explicit declaration to reserve memory space
- The declaration happens automatically when you assign a value to a variable
- The equal sign (=) is used to assign values to variables
  - counter = 100 # An integer assignment
  - miles = 1000.0 # A floating point
  - name = "John" # A string
  - a = b = c = 1 # Multiple assignments
  - a, b, c = 1, 2, "john" # Multiple assignments

Standard Data Types

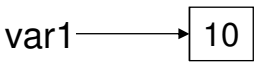
- Python has six data types
  - Numbers
  - String
  - List
  - Tuple
  - Dictionary
  - Set

Everything is an object

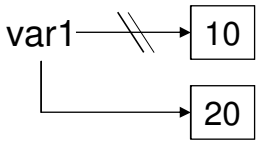
- Python data is represented by objects or by relations between objects
- Every object has an identity, a type and a value
  - Identity never changes once created location or address in memory
  - Type (e.g., integer, list) is unchangeable and determines the possible values it could have and operations that can be applied
  - Value of some objects is fixed (e.g., an integer) and can change for others (e.g., list)

Numbers

- Number data types store numeric values
- Immutable data types
  - changing the value of a number data type results in a newly allocated object
  - var1 = 10



  - var1 = 20



Numerical Types

- int (signed integers)
  - Positive or negative whole numbers with no decimal point
  - Integers in Python 3 are of unlimited size
- float (floating point real values)
  - Represent real numbers and are written with a decimal point
  - Can be in scientific notation (2.5e2 = 2.5 x 10^2 = 250)
- complex (complex numbers)
  - a + bJ form, where a and b are floats and J (or j) represents the square root of -1 (which is an imaginary number)
  - The real part of the number is a, and the imaginary part is b (rarely used)

Number examples

int	float	complex
10	0.0	3.14j
100	15.20	45.j
-786	-21.9	9.322e-36j
0b110	32.3+e18	.876j
-0o40	-90.	-.6545+0J
-0×260	-32.54e100	3e+26J
0×69	70.2-E12	4.53e-7j

Number Type Conversion

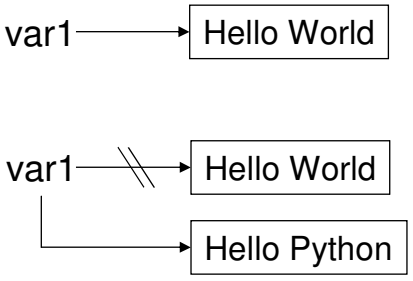
- Python converts numbers internally in an expression containing mixed types to a common type for evaluation
- Convert a number explicitly from one type to another to satisfy the requirements of an operator or function parameter
  - int(x) to convert x to a plain integer
  - float(x) to convert x to a floating-point number
  - complex(x) to convert x to a complex number with real part x and imaginary part zero
  - complex(x, y) to convert x and y to a complex number with real part x and imaginary part y. x and y are numeric expressions

Strings

- The most popular types in Python
- Create them by enclosing characters in quotes
  - Python treats single quotes the same as double quotes
  - var1 = 'Hello World!'
  - var2 = "Python Programming"
- To access substrings, use the square brackets for slicing along with the index or indices to obtain substring
  - print ("var1[0]: ", var1[0]) # 'H'
  - print ("var2[1:5]: ", var2[1:5]) # 'ytho'

Updating Strings

- You can "update" an existing string by (re)assigning a variable to another string
- The new value can be related to its previous value or to a completely different string altogether
  - var1 = 'Hello World'
  - var1 = var1[:6] + 'Python'
  - print(var1)



- In Python 3, all strings are represented in Unicode

Escape Characters

Escape Sequence	Meaning
\newline	Backslash and newline ignored
\\	Backslash (\)
\'	Single quote (')
\"	Double quote (")
\a	ASCII Bell (BEL)
\b	ASCII Backspace (BS)
\f	ASCII Formfeed (FF)
\n	ASCII Linefeed (LF)
\r	ASCII Carriage Return (CR)
\t	ASCII Horizontal Tab (TAB)
\v	ASCII Vertical Tab (VT)
\ooo	Character with octal value *ooo*
\xhh	Character with hex value *hh*

String Special Operators

- Assume string variable a holds 'Hello' and variable b holds 'Python'

Operator	Description	Example
+	Concatenation	a + b will give HelloPython
*	Repetition	a*2 will give -HelloHello
[]	Slice	a[1] will give e
[:]	Range Slice	a[1:4] will give ell
in	Membership	H in a will give 1
not in	Membership	M not in a will give 1
r/R	Raw String - Suppresses actual meaning of Escape characters. The "r" can be lowercase (r) or uppercase (R) and must be placed immediately preceding the first quote mark	print r'\n' : prints \n print R'\n' : prints \n

Triple Quotes

- Python's triple quotes comes to the rescue by allowing strings to span multiple lines, including NEWLINEs, TABs, and any other special characters
- The syntax for triple quotes consists of three consecutive single or double quotes.
  - para_str = """this is a long string that is made up of several lines and non-printable characters such as TAB (\t) and they will show up that way when displayed. You can explicitly given NEWLINE like this within the brackets [\n], or just a NEWLINE within the variable assignment will also show up."""
  - print (para_str)

Built-in string methods (1)

- Python includes a number of built-in string methods that are incredibly useful for string manipulation
  - Note that these return the modified string value; we cannot change the string's value in place because they're immutable!
- s.upper()
  - Converts all lowercase letters in string to uppercase
- s.lower()
  - Converts all uppercase letters in string to lowercase

Built-in string methods (2)

- s.isalpha(), s.isdigit(), s.isalnum(), s.isspace()
  - return True if string s is composed of alphabetic characters, digits, either alphabetic and/or digits, and entirely whitespace characters, respectively
- s.islower(), s.isupper()
  - return True if string s is all lowercase and all uppercase, respectively
- s.split([sep[, maxsplit]])
  - Split string into a list of substrings. The sep argument indicates the delimiting string (defaults to consecutive whitespace). The maxsplit argument indicates the maximum number of splits to be done (default is -1)

Built-in string methods (3)

- s.rsplit([sep[, maxsplit]])
  - Split string into a list of substrings, starting from the right
- s.strip([chars])
  - Return a copy of the string with leading and trailing characters removed. The chars string specifies the set of characters to remove (default is whitespace)
- s.rstrip([chars])
  - Return a copy of the string with only trailing characters removed

Built-in string methods (4)

- >>> "Python programming is fun!".split()
  ['Python', 'programming', 'is', 'fun!']
- >>> "555-867-5309".split('-')
  ['555', '867', '5309']
- >>> "***Python programming is fun***".strip('*')
  'Python programming is fun'

Ch.03 Python Operators

Types of Operator

- Python language supports the following types of operators
  - ◆Arithmetic Operators
  - ◆Comparison (Relational) Operators
  - ◆Assignment Operators
  - ◆Logical Operators
  - ◆Membership Operators
  - ◆Identity Operators
  - ◆Bitwise Operators

Arithmetic Operators

Operator	Description	Examples
+	Add numbers	x = a + b
-	Subtract numbers	x = a – b
*	Multiply numbers	x = a * b
/	Divide numbers	x = a / b
%	Divide two numbers and returns reminder	x = a % b then X will be X=4
**	Performs exponential (power) calculation on operators	x = a ** 2 then X will be 81
//	Floor Division - get an integer result, discarding any fractional result. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity)	9 // 2 = 4 and 9.0 // 2.0 = 4.0, -9 // 2 = -5, -9.0 // 2 = -5.0

assume a=9,b=5

Comparison Operators (1)

Operator	Name	Examples
==	Equal equal to	x == 2 (returns true)
!=	Not equal to	x != 2 (returns false)
<	Less than	x < 5 (returns true)
>	Greater than	x > 5 (returns false)
<=	Less than equal to	x <= 2 (returns true)
>=	Greater than equal to	x >= 2 (returns true)

assume x=2

Comparison Operators (2)

- Comparisons can be chained arbitrarily
- x < y <= z is equivalent to x < y and y <= z, except that y is evaluated only once
  - but in both cases z is not evaluated at all when x < y is found to be false
- Example:
  - >>> 4 <= 6 > 7
    False
  - >>> 5 < 6 > 3
    True
  - >>> 5 > 6 > 3
    False

Assignment Operators

Operator	Description	Example
=	Simple assignment	C = A + B assigns value of A + B into C
+=	Add AND assignment	C += A is equivalent to C = C + A
-=	Subtract AND assignment	C -= A is equivalent to C = C - A
*=	Multiply AND assignment	C *= A is equivalent to C = C * A
/=	Divide AND assignment	C /= A is equivalent to C = C / A
%=	Modulus AND assignment	C %= A is equivalent to C = C % A
**=	Exponent AND assignment	C **= A is equivalent to C = C ** A
//=	Floor Division AND assignment	C //= A is equivalent to C = C // A

Logical Operators (1)

Operator	Description	Example
and	If both the operands are non zero then condition becomes true	(A and B) is False
or	If any of the two operands is non zero then condition becomes true	(A or B) is True
not	Reverse the logical state of its operand. If a condition is true then Logical NOT operator will make false	not (A and B) is True

assume A=True, B=False

Logical Operators (2)

Operator	Description	Note
x or y	if x is false, then y, else x	(1)
x and y	if x is false, then x, else y	(2)
not x	if x is false, then True, else False	(3)

Notes:
(1) This is a short-circuit operator, so it only evaluates the second argument if the first one is false.
(2) This is a short-circuit operator, so it only evaluates the second argument if the first one is true.
(3) not has a lower priority than non-Boolean operators, so not a == b is interpreted as not (a == b), and a == not b is a syntax error.

Membership Operators

- Test for membership in a sequence, such as strings, lists, or tuples

Operator	Description	Example
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise	x in y, result is True
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise	x not in y, result is False

assume x=3, y=[1,2,3,4,5]

Identity Operators

- Python built-in function id() returns a unique integer as identity of object. Identity operators compare the memory locations of two objects

Operator	Description	Example
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise	x is y, results in True if id(x) is equals id(y).
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise	x is not y, here is not results in True if id(x) is not equal to id(y)

assume x=20, y=20

Bitwise Operators (1)

- Bitwise operator works on bits and performs bit-by-bit operation

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Bitwise Operators (2)

Operator	Description	Example
&	Binary AND Operator	(A & B) = 12, which is 0000 1100
|	Binary OR Operator	(A | B) = 61, which is 0011 1101
^	Binary XOR Operator	(A ^ B) = 49, which is 0011 0001
~	Binary Ones Complement Operator	(~A) = 61, which is 1100 0011 in 2's complement due to a signed binary number.
<<	Binary Left Shift Operator	A << 2 = 240, which is 1111 0000
>>	Binary Right Shift Operator	A >> 2 = 15, which is 0000 1111

assume A=60,B=13

Operator precedence (1)

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

Operator precedence (2)

Operator	Description
<<, >>	Shifts
+, -	Addition and subtraction
*, @, /, //, %	Multiplication, matrix multiplication division, remainder
+x, -x, ~x	Positive, negative, bitwise NOT
**	Exponentiation
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

Ch.04 Formatting, Lists, Tuples, Dictionary and Sets

String Formatting

String Formatting

- Python provide powerful built-in string formatters without requiring any additional libraries
- Two types of string formatters
  - ◆str.format()
    - New style
    - A large degree of flexibility and customization
  - ◆String Formatting Operator
    - Old style
    - Based on C printf style formatting that handles a narrower range of types

str.format() Method

- String formatting is accomplished via a built-in method of string objects
  - ◆ str.format(*args, **kwargs)
  - ◆ *args argument indicates that format accepts a variable number of positional arguments
  - ◆ **kwargs indicates that format accepts a variable number of keyword arguments
- The string can contain literal text or replacement fields delimited by braces {}. Each replacement field contains either the numeric index of a positional argument, or the name of a keyword argument
- A copy of the string is returned where each replacement field is replaced with the string value of the corresponding argument

String Formatting Operator

- Use string formatting or interpolation operator (% operator)
  - format % values
  - format is a string or Unicode object
  - % conversion specifications in format are replaced with zero or more elements of values
- The effect is similar to the using sprintf() in the C language
- If format requires a single argument, values may be a single non-tuple object. Otherwise, values must be a tuple with exactly the number of items specified by the format string, or a single mapping object (for example, a dictionary)

Basic formatting

- Simple positional formatting is probably the most common use-case
- Use it if the order of your arguments is not likely to change and you only have very few elements you want to concatenate

- Old
  '%s' % 'one'
  New
  '{}'.format('one')
  output
  'one'
- Old
  '%d %d' % (1, 2)
  New
  '{} {}'.format(1, 2)
  output
  '1 2'
- New
  '{1} {0}'.format('one', 'two')
  output
  two one

Numbers

- Of course it is also possible to format numbers
- Numbers can also be constrained to a specific width

- Old
  '%d' % 42
  New
  '{:d}'.format(42)
  output
  '42'
- Old
  '%f' % 3.141592653589
  New
  '{:f}'.format(3.141592653589)
  output
  '3.141593'
- Old
  '%4d' % 42
  New
  '{:4d}'.format(42)
  output
  ' 42'
- Old
  '%06.2f' % 3.141592653589
  New
  '{:06.2f}'.format(3.141592653589)
  output
  '003.14'

Datetime

- New style formatting also allows objects to control their own rendering
  - from datetime import datetime
  - New
    '{:%Y-%m-%d
    %H:%M}'.format(datetime(2001, 2, 3, 4, 5))
    output
    2001-02-03 04:05

String formatting (1)

```
>>> '{0}, {1}, {2}'.format('a', 'b', 'c')
'a, b, c'
>>> '{}, {}, {}'.format('a', 'b', 'c')
'a, b, c'
>>> '{2}, {1}, {0}'.format('a', 'b', 'c')
'c, b, a'
>>> '{2}, {1}, {0}'.format(*'abc')  # unpack
'c, b, a'
>>> '{0}{1}{0}'.format('abra', 'cad')
'abracadabra'
```


String formatting (2)

```
>>> 'Coords: {lat},
{long}'.format(lat='37.24N', long='-
115.81W')
'Coords: 37.24N, -115.81W'
>>> coord = {'lat': '37.24N', 'long': '-
115.81W'}
>>> 'Coords: {lat}, {long}'.format(**coord)
'Coords: 37.24N, -115.81W'
```

Lists, Tuples, Dictionary and Sets

Lists

- The list is the most versatile datatype available in Python
- Comma-separated values (items) between square brackets
- Items in a list need not be of the same type
  - list1 = ['physics', 'chemistry', 1997, 2000]
  - list2 = [1, 2, 3, 4, 5]
  - list3 = ["a", "b", "c", "d"]
- Similar to string indices, list indices start at 0, and lists can be sliced, concatenated and so on

Accessing Values in Lists

- To access values in lists, use the square brackets for slicing along with the index or indices to obtain value available at that index
  - list1 = ['physics', 'chemistry', 1997, 2000]
  - list2 = [1, 2, 3, 4, 5, 6, 7]
  - print ("list1[0]: ", list1[0]) # physics
  - print ("list2[1:5]: ", list2[1:5]) # [2, 3, 4, 5]

Updating Lists

- You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator, and you can add to elements in a list with the append() method
  - lst = ['physics', 'chemistry', 1997, 2000]
  - print ("Value available at index 2 : ", lst[2])
  - lst[2] = 2001
  - print ("New value available at index 2 : ", lst[2])

Basic List Operations

Python Expression	Results	Description
len([1, 2, 3])	3	Length
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	Concatenation
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	Repetition
3 in [1, 2, 3]	True	Membership
for x in [1,2,3] : print (x,end = ' ')		

Indexing, Slicing and Matrixes

- Assuming the following input
  - mylist = ['C++", 'Java', 'Python']

Python Expression	Results	Description
mylist[2]	'Python'	Offsets start at zero
mylist[-2]	'Java'	Negative: count from the right
mylist[1:]	['Java', 'Python']	Slicing fetches sections

Built-in List Functions

- len(list)
  - ◆Gives the total length of the list
- max(list)
  - ◆Returns item from the list with max value
- min(list)
  - ◆Returns item from the list with min value
- list(seq)
  - ◆Converts a tuple into list

Built-in List Methods

- list.append(obj)
  - ◆ Appends object obj to list
- list.count(obj)
  - ◆ Returns count of how many times obj occurs in list
- list.index(obj)
  - ◆ Returns the lowest index in list that obj appears
- list.insert(index, obj)
  - ◆ Inserts object obj into list at offset index
- list.remove(obj)
  - ◆ Removes object obj from list
- list.sort([func])
  - ◆ Sorts objects of list, use compare func if given

Tuples

- A tuple is a sequence of immutable Python objects
- Tuples are sequences, just like lists
- 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“
  - tup4 = () # empty tuple
  - tup5 = (50,) # single value tuple

Accessing Values in Tuples

- To access values in tuple, use the square brackets for slicing along with the index or indices to obtain the value available at that index
  - tup1 = ('physics', 'chemistry', 1997, 2000)
  - tup2 = (1, 2, 3, 4, 5, 6, 7)
  - print ("tup1[0]: ", tup1[0]) # physics
  - print ("tup2[1:5]: ", tup2[1:5]) # (2, 3, 4, 5)

Updating Tuples

- Tuples are immutable, which means you cannot update or delete the values of tuple elements. You are able to take portions of the existing tuples to create new tuples
  - tup1 = (12, 34.56)
  - tup2 = ('abc', 'xyz')
  - # Following action is not valid for tuples
tup1[0] = 100;
  - # You can create a new tuple as follows
tup3 = tup1 + tup2
  - print (tup3)

Basic Tuples Operations

Python Expression	Results	Description
len((1, 2, 3))	3	Length
(1, 2, 3) + (4, 5, 6)	(1, 2, 3, 4, 5, 6)	Concatenation
('Hi!',) * 4	('Hi!', 'Hi!', 'Hi!', 'Hi!')	Repetition
3 in (1, 2, 3)	True	Membership
for x in (1,2,3) : print (x, end = ' ')	1 2 3	Iteration

Indexing, Slicing, and Matrixes

- Indexing and slicing work the same way as strings, assuming the following input
  - tp=('C++', 'Java', 'Python')

Python Expression	Results	Description
tp[2]	'Python'	Offsets start at zero
tp[-2]	'Java'	Negative: count from the right
tp[1:]	('Java', 'Python')	Slicing fetches sections

Built-in Tuple Functions

- len(tuple)
  - Gives the total length of the tuple
- max(tuple)
  - Returns item from the tuple with max value
  - Elements must be the same data type
- min(tuple)
  - Returns item from the tuple with min value
  - Elements must be the same data type
- tuple(seq)
  - Converts a list into tuple
  - tuple([1,2,3])

Dictionary

- Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces
- An empty dictionary without any items is written with just two curly braces, {}
- Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples

Accessing Values in Dictionary

- To access dictionary elements, you can use the familiar square brackets along with the key to obtain its value
  - dict1 = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
  - print ("dict1['Name']: ", dict1['Name'])
  - print ("dict1['Age']: ", dict1['Age'])
- If we attempt to access a data item which is not a part of the dictionary, we get an KeyError exception
  - print("dict1['Weight']: ", dict1['Weight'])

Updating Dictionary

- You can update a dictionary by adding a new entry or a key-value pair, modifying an existing entry, or deleting an existing entry
  - dict1 = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
  - dict1['Age'] = 8; # update existing entry
  - dict1['School'] = "DPS School" # Add new entry
  - print ("dict1['Age']: ", dict1['Age'])
  - print ("dict1['School']: ", dict1['School'])

Delete Dictionary Elements

- You can either remove individual dictionary elements or clear the entire contents of a dictionary. You can also delete entire dictionary in a single operation.
  - dict1 = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
  - del dict1['Name'] # remove entry with key 'Name'
  - dict1.clear() # remove all entries in dict
  - del dict1 # delete entire dictionary

Built-in Dictionary Functions

- len(dict)
  - Gives the total length of the dictionary. This would be equal to the number of items in the dictionary
- str(dict)
  - Produces a printable string representation of a dictionary
- type(variable)
  - Returns the type of the passed variable. If passed variable is dictionary, then it would return a dictionary type

Built-in Dictionary Methods (1)

- dict.clear()
  - Removes all elements of dictionary dict
- dict.copy()
  - Returns a shallow copy of dictionary dict
- dict.fromkeys()
  - Create a new dictionary with keys from seq and values set to value
- dict.get(key, default=None)
  - For key key, returns value or default if key not in dictionary
- dict.has_key(key)
  - Removed, use the in operation instead.

Built-in Dictionary Methods (2)

- dict.items()
  - Returns a list of dict's (key, value) tuple pairs
- dict.keys()
  - Returns list of dictionary dict's keys
- dict.setdefault(key, default = None)
  - Similar to get(), but will set dict[key] = default if key is not already in dict
- dict.update(dict2)
  - Adds dictionary dict2's key-values pairs to dict
- dict.values()
  - Returns list of dictionary dict's values

Sets

- A set is an unordered collection with no duplicate elements
- Basic uses include membership testing and eliminating duplicate entries
- Set objects also support mathematical operations like union, intersection, difference, and symmetric difference
- Curly braces or the set() function can be used to create sets
  - Note: to create an empty set you have to use set(), not {}; the latter creates an empty dictionary

Types of Sets

- set
  - The set type is mutable, the contents can be changed using methods like add() and remove()
  - Since it is mutable, it has no hash value and cannot be used as either a dictionary key or as an element of another set
- frozenset
  - The frozenset type is immutable and hashable, its contents cannot be altered after it is created
  - It can therefore be used as a dictionary key or as an element of another set

Using Sets

- basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
- print(basket) # duplicates are removed
  - {'orange', 'banana', 'pear', 'apple'}
- 'orange' in basket # fast membership testing
  - True
- 'crabgrass' in basket
  - False

Basic Sets Operations

Python Expression	Results	Description
len({1, 2, 3})	3	Length
3 in {1, 2, 3}	True	Membership
for x in {1,5,2,3} : print (x, end = ' ')	1 2 3 5	Iteration (unordered)
{1,2,3} | {2,3,4}	{1, 2, 3, 4}	Union
{1,2,3} & {2,3,4}	{2, 3}	Intersection
{1,2,3} - {2,3,4}	{1}	Difference
{1,2,3} ^ {2,3,4}	{1, 4}	Symmetric difference
{1,2} < {1,2,3,4} {1,2} > {1,2,3,4}	True False	Subset Superset

Built-in Sets Functions

- len(set)
  - ◆Gives the total length of the set
- max(set)
  - ◆Return the largest item in the set
- min(set)
  - ◆Returns the smallest item in the set
- sum(set)
  - ◆Sums the items of the set and returns the total

Built-in Sets Methods (1)

- s.isdisjoint(other)
  - Return True if the set has no elements in common with other. Sets are disjoint if and only if their intersection is the empty set
- s.issubset(other)
  - Test whether every element in the set is in other
- s.issuperset(other)
  - Test whether every element in other is in the set
- s.union(*others)
  - Return a new set with elements from the set and all others

Built-in Sets Methods (2)

- s.intersection(*others)
  - Return a new set with elements common to the set and all others
- s.difference(*others)
  - Return a new set with elements in the set that are not in the others
- s.symmetric_difference(other)
  - Return a new set with elements in either the set or other but not both
- s.copy()
  - Return a new set with a shallow copy of s

Built-in Sets Update Methods (1)

- s.update(*others)
  - set |= other | ...
  - Update the set, adding elements from all others
- s.intersection_update(*others)
  - set &= other & ...
  - Update the set, keeping only elements found in it and all others
- s.difference_update(*others)
  - set -= other | ...
  - Update the set, removing elements found in others
- s.symmetric_difference_update(other)
  - set ^= other
  - Update the set, keeping only elements found in either set, but not in both

Built-in Sets Update Methods (2)

- s.add(elem)
  - Add element elem to the set
- s.remove(elem)
  - Remove element elem from the set
  - Raises KeyError if elem is not contained in the set
- s.discard(elem)
  - Remove element elem from the set if it is present
- s.pop()
  - Remove and return an arbitrary element from the set
  - Raises KeyError if the set is empty
- s.clear()
  - Remove all elements from the set

Ch.05 Decision Making and Loops

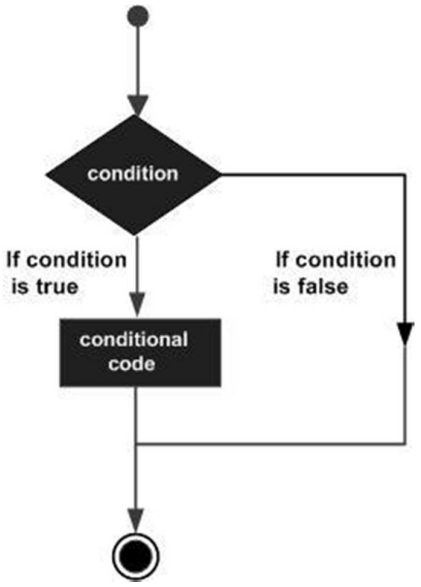
Decision Making

Python Decision Making

Statement	Description
if statements	An if statement consists of a boolean expression followed by one or more statements
if...else statements	An if statement can be followed by an optional else statement, which executes when the boolean expression is FALSE
nested if statements	You can use one if or else if statement inside another if or else if statement(s)

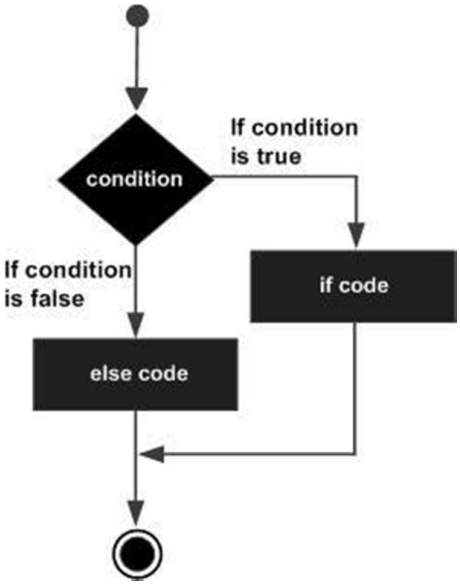
if Statement

- Syntax
  - if expression:
    statement(s)
- Example
  - var1 = 100
    if var1:
    print ("1 true expression")
    print (var1)
    var2 = 0
    if var2:
    print ("2 true expression")
    print (var2)
    print ("Good bye!")



if...else Statements

- Syntax
  - if expression:
      statement(s)
    else:
      statement(s)
- Example
  - if amount<1000:
      discount = amount*0.05
      print("Discount",discount)
    else:
      discount = amount*0.10
      print("Discount",discount)



if...elif...else Statements (1)

- syntax
  - if expression1:
      statement(s)
    elif expression2:
      statement(s)
    elif expression3:
      statement(s)
    else:
      statement(s)

if...elif...else Statements (2)

- Example
  - amount = int(input("Enter amount: "))

```
if amount<1000:
    discount = amount*0.05
    print ("Discount",discount)
elif amount<5000:
    discount = amount*0.10
    print ("Discount",discount)
else:
    discount = amount*0.15
    print ("Discount",discount)

print ("Net payable:",amount-discount)
```


Nested if Statements (1)

- Syntax
  -

```
if expression1:
    statement(s)
    if expression2:
        statement(s)
    elif expression3:
        statement(s)
    else
        statement(s)
elif expression4:
    statement(s)
else:
    statement(s)
```

Nested if Statements (2)

- Example
  - num = int(input("enter number"))

```
if num%2 == 0:
    if num%3 == 0:
        print ("Divisible by 3 and 2")
    else:
        print ("divisible by 2 not divisible by 3")
else:
    if num%3 == 0:
        print ("divisible by 3 not divisible by 2")
    else:
        print  ("not Divisible by 2 not divisible by 3")
```


Single Statement Suites

- If the suite of an if clause consists only of a single line, it may go on the same line as the header statement
- Example
  - var = 100

```
if ( var  == 100 ) : print ("Value is 100")
print ("Good bye!")
```

Conditional expressions

- Sometimes called a “ternary operator”
- Have the lowest priority of all Python operations
- Syntax
  - x if C else y
  - first evaluates the condition C rather than x. If C is true, x is evaluated and its value is returned; otherwise, y is evaluated and its value is returned

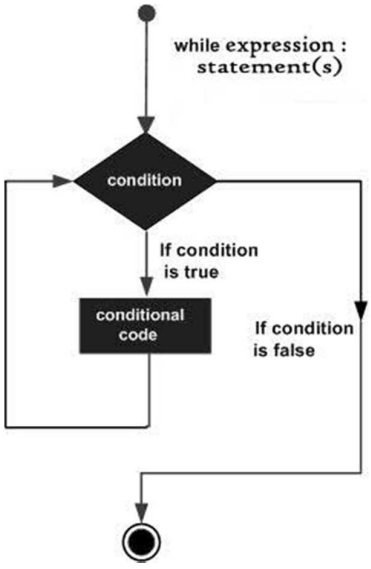
Loops

Python Loop Statements

Statement	Description
while loop	Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body
for loop	Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable
nested loops	You can use one or more loop inside any another while, or for loop

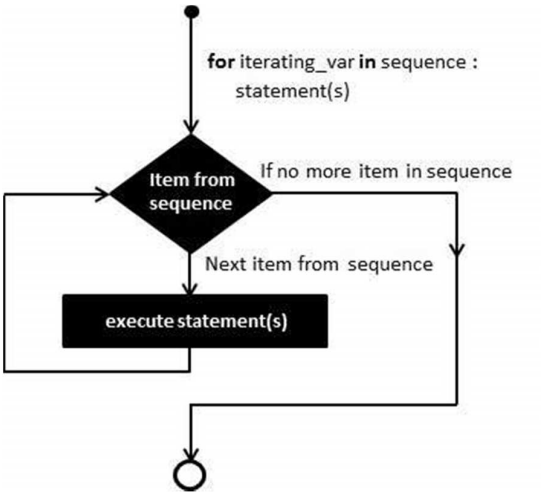
while Loop Statements

- Syntax
  - while expression:
    statement(s)
- Example
  - count = 0
    while (count < 9):
    print ('count is:', count)
    count = count + 1
    print ("Good bye!")
- Loop might not ever run



for Loop Statements (1)

- Syntax
  - for iterating_var in sequence:
    statements(s)
- The built-in range() function generates an iterator of arithmetic progressions
- Example
  - for var in list(range(5)):
    print(var)



for Loop Statements (2)

- Example
  - # traversal of a string sequence

```
for letter in 'Python':
    print ('Current Letter :', letter)
print()
fruits = ['banana', 'apple',  'mango']
# traversal of List sequence
for fruit in fruits:
    print ('Current fruit :', fruit)
print ("Good bye!")
```

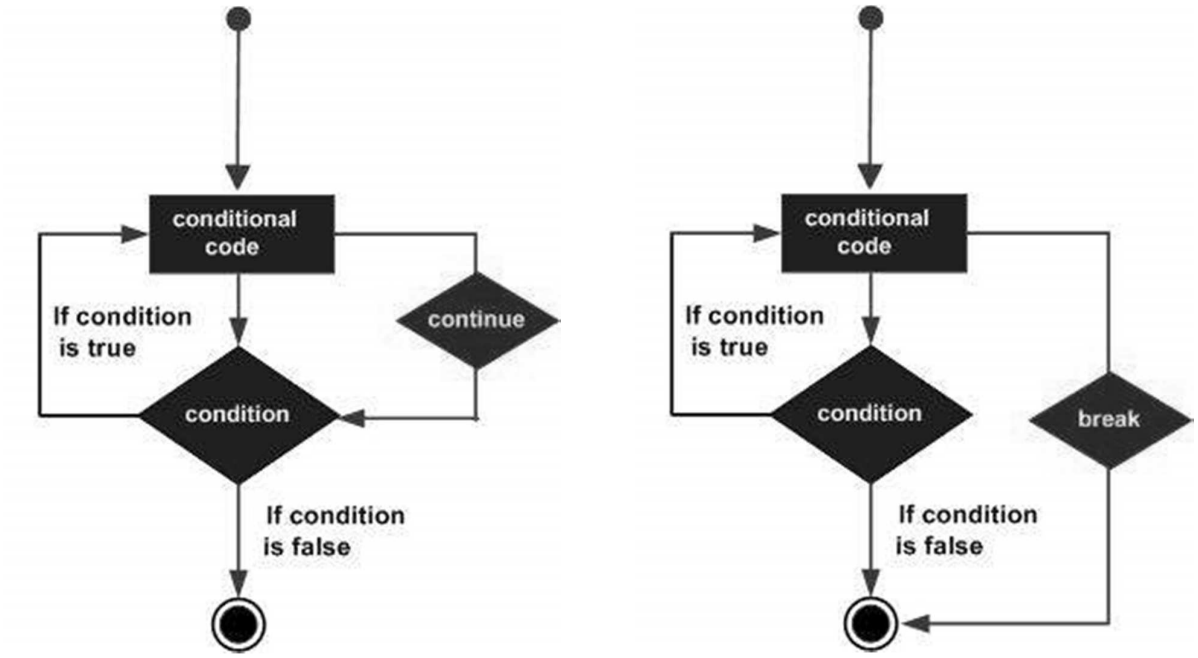
Iterating by Sequence Index

- An alternative way of iterating through each item is by index offset into the sequence itself
- Example
  - fruits = ['banana', 'apple', 'mango']
    for index in range(len(fruits)):
        print ('Current fruit :', fruits[index])
    print ("Good bye!")

break and continue Statements (1)

- The break statement, like in C, breaks out of the smallest enclosing for or while loop
- The continue statement, also borrowed from C, continues with the next iteration of the loop

break and continue Statements (2)



else Statement with Loops (1)

- Python supports having an else statement associated with a loop statement
- If the else statement is used with a for loop, the else block is executed only if for loops terminates normally (and not by encountering break statement)
- If the else statement is used with a while loop, the else statement is executed when the condition becomes false

else Statement with Loops (2)

- Example
  - numbers =
    [11,33,55,39,55,75,37,21,23,41,13]
    for num in numbers:
        if num%2 == 0:
            print ('the list contains an even number')
            break
    else:
        print ('the list has no even number')

Nested loops (1)

- Syntax
  - for iterating_var in sequence:
        for iterating_var in sequence:
            statements(s)
        statements(s)
  - while expression:
        while expression:
            statement(s)
        statement(s)

Nested loops (2)

- Example
  - ```
    import sys
    for i in range(1,11):
        for j in range(1,11):
            k=i*j
            print (k, end=' ')
        print()
    ```
  - ```
    1 2 3 4 5 6 7 8 9 10
    2 4 6 8 10 12 14 16 18 20
    …
    8 16 24 32 40 48 56 64 72 80
    9 18 27 36 45 54 63 72 81 90
    10 20 30 40 50 60 70 80 90 100
    ```


List Comprehensions (1)

- Python supports a concept called "list comprehensions". It can be used to construct lists in a very natural, easy way, like a mathematician is used to do
- In mathematics
  - $S = \{x^2 : x \text{ in } \{0 \ldots 9\}\}$
  - $V = (1, 2, 4, 8, \ldots, 2^{10})$
  - $M = \{x \mid x \text{ in } S \text{ and } x \text{ even}\}$

List Comprehensions (2)

- In Python
  - >>> S = [x**2 for x in range(10)]
    >>> S
    [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
  - >>> V = [2**i for i in range(10)]
    >>> V
    [1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
  - >>> M = [x for x in S if x % 2 == 0]
    >>> M
    [0, 4, 16, 36, 64]

List Comprehensions (3)

- >>> words = 'This is a book'.split()
  >>> words
  ['This', 'is', 'a', 'book']
- >>> stuff = [[w.upper(), len(w)] for w in words]
  >>> for i in stuff:
  ... print(i)
  ...
  ['THIS', 4]
  ['IS', 2]
  ['A', 1]
  ['BOOK', 4]
- >>> {x:x*x for x in range(5)}
  {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

Ch.06 Python Functions and Generators

Functions

Functions

- A function is a block of organized, reusable code that is used to perform a single, related action
- Functions provide better modularity for your application and a high degree of code reusing
- Python provides many built-in functions like print(), etc. but you can also create your own functions. These functions are called user-defined functions

Defining a Function

- Function blocks begin with the keyword “def” followed by the function name and parentheses ( )
- You can define parameters inside these parentheses
- The code block within every function starts with a colon (:) and is indented
- The statement “return [expression]” exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as “return None”

Defining a Function

- Syntax
  - def functionname(parameters):
    function_suite
    return [expression]
- Example
  - def printme(str):
    print (str)
    return

Calling a Function

- Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt
  - # Function definition is here
    def printme(str):
    "This prints a passed string into this function"
    print (str)
    return
    # Now you can call printme function
    printme("first call!")
    printme("Again second call")

Pass by Reference vs Value

- All parameters (arguments) in the Python language are passed by reference
- If you change the mutable data which a parameter refers to within a function, the change also reflects back in the calling function

Pass by Reference vs Value

- Example
  - # Function definition is here

```
def changeme( mylist ):
    print ("Values in the function before change: ", mylist)
    mylist[2]=50
    print ("Values in the function after change: ", mylist)
    return
# Now you can call changeme function
mylist = [10,20,30]
changeme( mylist )
print ("Values outside the function: ", mylist)
```

Pass by Reference vs Value

- Other example
  - ```
    # Function definition is here
    def changeme( mylist ):
        mylist = [1,2,3,4]
        # This would assign new reference in mylist
        print ("Values in the function: ", mylist)
        return
    # Now you can call changeme function
    mylist = [10,20,30]
    changeme( mylist )
    print ("Values outside the function: ", mylist)
    ```
  - The parameter mylist is local to the function changeme. Changing mylist within the function does not affect mylist outside the function

Function Arguments

- You can call a function by using the following types of formal arguments
  - Required arguments
  - Keyword arguments
  - Default arguments
  - Variable-length arguments
    ```

Required Arguments

- Required arguments are the arguments passed to a function in correct positional order
- The number of arguments in the function call should match exactly with the function definition
- Example
  - # Function definition is here
    def printme(str):
        print (str)
        return
    # Now you can call printme function
    printme("test")

Keyword Arguments

- Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name
- This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters

Keyword Arguments

- Example
  - # Function definition is here
    def printme(str):
        print (str)
        return
    # Now you can call printme function
    printme(str = "My string")
- Other example
  - # Function definition is here
    def printinfo(name, age):
        print ("Name: ", name)
        print ("Age ", age)
        return
    # Now you can call printinfo function
    printinfo(age = 50, name = "miki")

Default Arguments

- A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument
- Example
  - # Function definition is here
    def printinfo(name, age = 35):
        print ("Name: ", name)
        print ("Age ", age)
        return
    # Now you can call printinfo function
    printinfo(age = 50, name = "miki") # age is 50
    printinfo(name = "miki") # age is 35

Variable-length Arguments

- You may need to process a function for more arguments than you specified while defining the function
- These arguments are called variable-length arguments and are not named in the function definition, unlike required and default arguments
- An asterisk (*) is placed before the variable name that holds the values of all nonkeyword variable arguments
  - This tuple remains empty if no additional arguments are specified during the function call

Variable-length Arguments

- Example
- # Function definition is here

```
def printinfo( arg1, *vartuple ):
    print ("Output is: ")
    print (arg1)
    for var in vartuple:
        print (var)
    return
# Now you can call printinfo function
printinfo( 10 )
printinfo( 70, 60, 50 )
```

The Anonymous Functions

- The anonymous functions are not declared in the standard manner by using the “def” keyword
  - Use the “lambda” keyword to create small anonymous functions
- Lambda forms can take any number of arguments but return just one value in the form of an expression
  - They cannot contain commands or multiple expressions

The Anonymous Functions

- An anonymous function cannot be a direct call to print because lambda requires an expression
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace
- Although it appears that lambdas are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is for performance reason

The Anonymous Functions

- The syntax of lambda functions contains only a single statement
  - lambda [arg1 [,arg2,.....argn]]:expression
- Example
  - # Function definition is here
    sum = lambda arg1, arg2: arg1 + arg2
    # Now you can call sum as a function
    print ("Value of total : ", sum(10, 20)) # 30
    print ("Value of total : ", sum(20, 20)) # 40

The return Statement

- The statement return [expression] exits a function, optionally passing back an expression to the caller
  - A return statement with no arguments is the same as return None
- Example
  - # Function definition is here
    def sum(arg1, arg2):
        total = arg1 + arg2
        print ("Inside the function : ", total)
        return total
    # Now you can call sum function
    total = sum(10, 20)
    print ("Outside the function : ", total) # 30

Scope of Variables

- All variables in a program may not be accessible at all locations in that program
  - This depends on where you have declared a variable
- The scope of a variable determines the portion of the program where you can access a particular identifier
- There are two basic scopes of variables in Python
  - Global variables
  - Local variables

Global vs. Local variables

- Variables that are defined inside a function body have a local scope, and those defined outside have a global scope
- Local variables can be accessed only inside the function in which they are declared
- Global variables can be accessed throughout the program body by all functions

Global vs. Local variables

- Example
  - total = 0 # This is global variable
    # Function definition is here
    def sum(arg1, arg2):
        total = arg1 + arg2; # total is local variable
        # print 30
        print ("Inside the function local total : ", total)
        return total
    # Now you can call sum function
    sum(10, 20)
    # print 0
    print ("Outside the function global total : ", total)

The global Statement

- If you need to modify a global variable from within a function, use the “global” statement
- Example
  - def spam():
        global eggs
        eggs = 'spam' # this is the global
    def bacon():
        eggs = 'bacon' # this is a local
    def ham():
        print(eggs) # this is the global

    eggs = 42 # this is the global
    spam()
    print(eggs) # spam

Iterators/Generators

Iterators (1)

- The iterator is an abstraction, which enables the programmer to access all the elements of a container (a set, a list and so on) without any deeper knowledge of the data structure of this container object
- Iterators are a fundamental concept of Python
- Mostly, iterators are implicitly used, like in the for-loop of Python
  - fruits = ["apple", "orange", "banana", "grape"]
    for f in fruits:
        print("fruit: ", f)
  - We are iterating over a list, but you shouldn't be mistaken: A list is not an iterator, but it can be used like an iterator

Iterators (2)

- Internally, the for loop calls the “next” function and terminates, when it gets StopIteration
  - fruits = ["apple", "orange", "banana", "grape"]
    fruits_iterator = iter(fruits)
    next(fruits_iterator) # apple
    next(fruits_iterator) # orange
    next(fruits_iterator) # banana
    next(fruits_iterator) # grape
    next(fruits_iterator) # StopIteration exception

Generators (1)

- Generators are a special kind of function, which enable us to implement or generate iterators
- A generator is a function which returns a generator object by using yield statements
  - This generator object can be seen like a function which produces a sequence of results instead of a single object
- The yield statement turns a functions into a generator
- The execution of the code stops when a yield statement has been reached
  - The value behind the yield will be returned

Generators (2)

- At the next call, the execution continues with the statement following the yield statement and the variables have the same values as they had in the previous call
  - This means that all the local variables still exists, because they are automatically saved between calls
- There may be more than one yield statement in the code of a generator or the yield statement might be inside the body of a loop
- The iterator is finished, if the generator body is completely worked through or if the program flow encounters a return statement without a value

Generator Example (1)

```
# A simple generator function
def my_gen():
    n = 1
    print('This is printed first')
    # Generator contains yield statements
    yield n
    n += 1
    print('This is printed second')
    yield n
    n += 1
    print('This is printed at last')
    yield n

for item in my_gen():
    print(item)
```

Generator Example (2)

- a = my_gen()
- next(a)
  - This is printed first
  - 1
- next(a)
  - This is printed second
  - 2
- next(a)
  - This is printed at last
  - 3
- next(a)
  - Traceback (most recent call last):
  - ...
  - StopIteration

Ch.07 Files I/O and Exceptions

Files I/O

Printing to the Screen

- The simplest way to produce output is using the print function where you can pass zero or more expressions separated by commas
- Print function converts the expressions you pass into a string and writes the result to standard output
  - print ("Python is a great language,", " try it!")

Reading Keyboard Input

- input() function read data from keyboard
- eval() function return the result of the evaluated expression
- >>> print(input('What is your name? '))
  What is your name? John
  John
  >>> print(eval(input('Do some math: ')))
  Do some math: 2+2*5
  12

Opening and Closing Files

- Python provides basic functions and methods necessary to manipulate files by default
  - Python includes a file object that we can use to manipulate files
- Use the open() function
  - f = open(file_name [, access_mode][, buffering])
  - The first argument is the filename
  - The second argument accepts a few special characters
  - If the buffering value is set to 0, no buffering takes place. If the buffering value is 1, line buffering is performed while accessing a file
    - If you specify the buffering value as an integer greater than 1, then buffering action is performed with the indicated buffer size. If negative, the buffer size is the system default

File Access Modes

- The default mode is 'r' (open for reading text, synonym of 'rt')

Character	Meaning
'r'	open for reading (default)
'w'	open for writing, truncating the file first
'x'	open for exclusive creation, failing if the file already exists
'a'	open for writing, appending to the end of the file if it exists
'b'	binary mode
't'	text mode (default)
'+'	open a disk file for updating (reading and writing)

The file Object Attributes

- Once a file is opened and you have one file object, you can get various information related to that file
- file.closed
  - Returns true if file is closed, false otherwise.
- file.mode
  - Returns access mode with which file was opened.
- file.name
  - Returns name of the file
- Example
  - fo = open("foo.txt", "wb")
    print ("Name of the file: ", fo.name)
    print ("Closed or not : ", fo.closed)
    print ("Opening mode : ", fo.mode)
    fo.close()

The close() Method

- file.close()
  - Flushes any unwritten information and closes the file object, after which no more writing can be done
  - Python automatically closes a file when the reference object of a file is reassigned to another file
- Example
  - fo = open("foo.txt", "wb")
    print ("Name of the file: ", fo.name)
    # Close opened file
    fo.close()

The write() Method

- file.write()
  - Writes any string to an open file
  - It is important to note that Python strings can have binary data and not just text
  - Does not add a newline character ('\n') to the end of the string
- Example
  - fo = open("foo.txt", "w")
    fo.write("Python is a great language!!\n")
    # Close opend file
    fo.close()

The read() Method

- file.read()
  - Reads a string from an open file
  - It is important to note that Python strings can have binary data. apart from text data
- Example
  - fo = open("foo.txt", "r+")
    str = fo.read(10)
    print ("Read String is : ", str) # "Python is "
    # Close opened file
    fo.close()

Exceptions

Exceptions

- Errors that are encountered during the execution of a Python program are exceptions
- There are a number of built-in exceptions
  - https://docs.python.org/3.6/library/exceptions.html#bltinexceptions
- Python provides two very important features to handle any unexpected error in Python programs and to add debugging capabilities
  - Exception Handling
  - Assertions

Handling exceptions

- Use try/except blocks to catch and recover from exceptions
- Syntax
  - try:
      stmt
      ...
    except Exception1:
      If there is Exception1 then execute this block
    except Exception2:
      If there is Exception2, then execute this block
      ...
    else:
      If there is no exception then execute this block
    finally:
      This would always be executed

Handling exceptions

- First, the try clause (the statement(s) between the try and except keywords) is executed
- If no exception occurs, the except clause is skipped and execution of the try statement is finished
- If an exception occurs during execution of the try clause, the rest of the clause is skipped
  - Then if its type matches the exception named after the except keyword, the except clause is executed, and then execution continues after the try statement
- If an exception occurs which does not match the exception named in the except clause, it is passed on to outer try statements; if no handler is found, it is an unhandled exception and execution stops with an error message

Handling exceptions

- A try statement may have more than one except clause, to specify handlers for different exceptions
- At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding try clause
- An except clause may name multiple exceptions as a parenthesized tuple, for example:
  - except (RuntimeError, TypeError, NameError):
    pass

Handling exceptions

- The try/except clause options are as follows

Clause	Interpretation
except:	Catch all (or all other) exception types
except name:	Catch a specific exception only
except name as arg:	Catch the listed exception and its Instance
except (name1, name2):	Catch any of the listed exceptions
except (name1, name2) as arg:	Catch any of the listed exceptions and its instance
else:	Run if no exception is raised
finally:	Always perform this block

Exception Example

```
try:
    fh = open("testfile", "w“)
    fh.write(“Test file for exception
handling“)
except IOError:
    print ("Error: can\'t find file or read data“)
else:
    print ("Written to file successfully“)
    fh.close()
```


Exception Example

```
try:
    fh = open("testfile", "w“)
    try:
      fh.write(“test for exception handling“)
    finally:
      print ("Going to close the file“)
      fh.close()
except IOError:
    print ("Error: can\'t find file or read data")
```

Argument of an Exception

- An exception can have an argument, which is a value that gives additional information about the problem
  - The contents of the argument vary by exception
- Example
  - def data_convert(var):
        try:
            return int(var)
        except ValueError as e:
            print ("The data is not numbers\n", e)
    # Call above function here
    data_convert("xyz")

Exception Example

- while True:
    try:
        x = int(input("Enter a number: "))
    except ValueError:
        print("Not a valid number. Try again.")
    except (TypeError, IOError) as e:
        print(e)
    else:
        print("No errors encountered!")
    finally:
        print("Cleanup processing…")

Ch.08 Python Modules

Module

- A module allows to logically organize Python code. Grouping related code into a module makes the code easier to understand and use
- A module is a Python object with arbitrarily named attributes that you can bind and reference
- Simply, a module is a file consisting of Python code. A module can define functions, classes and variables. A module can also include runnable code

The import Statement

- You can use any Python source file as a module by executing an import statement in some other Python source file
- Syntax
  - import module1[, module2[,... moduleN]
- When the interpreter encounters an import statement, it imports the module if the module is present in the search path
- A module is loaded only once, regardless of the number of times it is imported. This prevents the module execution from happening repeatedly, if multiple imports occur

Define and Import Module

- The Python code for a module named “support” normally resides in a file named “support.py”
- support.py
  - def print_func(para):
        print ("Hello : ", para)
        return
- # Import module support
  - import support
    # call defined function
    support.print_func(“Rose")

Locating Modules

- When you import a module, the Python interpreter searches for the module in the following sequences
  - The current directory
  - If the module is not found, Python then searches each directory in the shell variable PYTHONPATH
  - If all else fails, Python checks the default path. On Linux, this default path is normally /usr/local/lib/python3/
- The module search path is stored in the system module sys as the sys.path variable

The from...import Statement (1)

- Python's from statement lets you import specific attributes from a module into the current namespace
- Syntax
  - from modname import name1[, name2[, ... nameN]]
- # fibo.py

```
def fib(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a + b
    return result
```

The from...import Statement (2)

- import fibo
  fibo.fib(10)
  - Import entire module
- from fibo import fib
  fib(10)
  - This statement does not import the entire module fibo into the current namespace; it just introduces the item fib from the module fibo into the global symbol table of the importing module

The from...import * Statement

- It is also possible to import all the names from a module into the current namespace by using the following import statement
  - from fibo import *
    fib(20)
- This provides an easy way to import all the items from a module into the current namespace; however, this statement should be used carefully

The dir( ) Function

- The dir() built-in function returns a sorted list of strings containing the names defined by a module
- The list contains the names of all the modules, variables and functions that are defined in a module
  - import math
    content = dir(math)
    print(content)
    ['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']

Special Variables

- __name__ is the module's name
  - __main__ : Top-level script environment
    - '__main__' is the name of the scope in which top-level code executes
  - A module's __name__ is set equal to '__main__' when read from standard input, a script, or from an interactive prompt
    - if __name__ == "__main__":
      # execute only if run as a script
      main()
- __file__ is the filename from which the module was loaded

The Python Standard Library

- A collection of modules that are distributed with every Python installation. It is a vast assortment of useful tools and interfaces, which covers a very wide range of domains
- Besides the standard library, there is also the Python Package Index (PyPI), the official third-party repository for everything from simple modules to elaborate frameworks written by other Python programmers. As of right now, there are 116,198 packages in PyPI (2017/09/05)
  - https://pypi.python.org/pypi

Standard Library Built-Ins

- A lot of data types – such as numbers and lists – which are part of the “core” of Python. That is, you don’t need to import anything to use them
- However, it’s the standard library that actually defines these types, as well as many other built-in components

Standard Library
Built-In Constants (1)

- There are a few built-in constants defined by the standard library:
  - True: true value of a bool type
  - False: false value of a bool type
- a = True
  b = False
  if a is True:
  print("a is true.“)
  else:
  print("a is false“)
  if b is True:
  print("b is true.“)
  else:
  print("b is false.“)

Standard Library
Built-In Constants (2)

- None: used to represent the absence of a value. Similar to the null keyword in many other languages
- conn = None
  try:
  database = MyDatabase(db_host, db_user, db_password, db_database)
  conn = database.connect()
  except DatabaseException:
  pass

  if conn is None:
  print('The database could not connect')
  else:
  print('The database could connect')

Standard Library
Built-In Functions

		Built-in Functions		
abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

Using Date and Time module

Using time Module

- A Python program can handle date and time in several ways. Converting between date formats is a common chore for computers. Python's time module help track dates and times
- The time module is responsible for providing time-related functions and conversion methods
  - import time

time Module

- Time intervals are floating-point numbers in units of seconds. Particular instants in time are expressed in seconds since 12:00am, January 1, 1970(epoch)
- The most commonly used methods are:
  - time.time() – returns the time in seconds since the epoch
  - time.sleep(s) – suspends execution for s seconds
  - time.clock() – returns the current processor time in seconds

Using time Methods

```
>>> import time
>>> def timer():
...     s = time.time()
...     time.sleep(5)
...     e = time.time()
...     print (e-s)
>>> def cpu_timer():
...     s = time.clock()
...     time.sleep(5)
...     e = time.clock()
...     print (e-s)
>>> timer()
5.00614309311
>>> cpu_timer()
0.000121
```


Using datetime Module

- if you want to display a date in a more convenient format, or do arithmetic with dates (for example, figuring out what date was 205 days ago or what date is 123 days from now), you should use the datetime module
- The datetime module has its own datetime data type. datetime values represent a specific moment in time
  - import datetime

Converting datetime Objects into Strings

- Use the strftime() method to display a datetime object as a string. (The *f* in the name of the strftime() function stands for format)
- The strftime() method uses directives similar to Python's string formatting
- >>> from datetime import datetime
  >>> oct21st = datetime(2015, 10, 21, 16, 29, 0)
  >>> oct21st.strftime('%Y/%m/%d %H:%M:%S')
  '2015/10/21 16:29:00'
  >>> oct21st.strftime('%I:%M %p')
  '04:29 PM'
  >>> oct21st.strftime("%B of '%y")
  "October of '15"

strftime() format codes

- strftime directive Meaning
  - %Y Year with century, as in '2014'
  - %y Year without century, '00' to '99' (1970 to 2069)
  - %m Month as a decimal number, '01' to '12'
  - %B Full month name, as in 'November'
  - %b Abbreviated month name, as in 'Nov'
  - %d Day of the month, '01' to '31'
  - %j Day of the year, '001' to '366'
  - %w Day of the week, '0' (Sunday) to '6' (Saturday)
  - %A Full weekday name, as in 'Monday'
  - %a Abbreviated weekday name, as in 'Mon'
  - %H Hour (24-hour clock), '00' to '23'
  - %I Hour (12-hour clock), '01' to '12'
  - %M Minute, '00' to '59'
  - %S Second, '00' to '59'
  - %p 'AM' or 'PM'

Standard library: sys

- The sys module allows you to receive information about the runtime environment as well as make modifications to it
- One of the most common ways to use the sys module is to access arguments passed to the program. This is done with the sys.argv list
- To use the sys module, just execute the import sys statement
  - The first element of the sys.argv list is always the module name, followed by the whitespace-separated arguments

Standard library: sys

- ```
  import sys
  for i in range(len(sys.argv)):
      print "sys.argv[" + str(i) + "] is " +
  sys.argv[i]

  $ python test.py one two three four
  sys.argv[0] : testargs.py
  sys.argv[1] : one
  sys.argv[2] : two
  sys.argv[3] : three
  sys.argv[4] : four
  ```
  ```

Standard library: sys

- The sys.path variable specifies the locations where Python will look for imported modules
- The sys.path variable is also a list and may be freely manipulated by the running program. The first element is always the “current” directory where the top-level module resides
  - import sys

    print("path has", len(sys.path), "members")
    print(sys.path)

Standard library: sys

- Use sys.builtin_module_names to see which modules are built-in
  - >>>import sys
    >>>sys.builtin_module_names
    ('_ast', '_codecs', '_collections', '_functools', '_imp', '_io', '_locale', '_operator', '_signal', '_sre', '_stat', '_string', '_symtable', '_thread', '_tracemalloc', '_warnings', '_weakref', 'atexit', 'builtins', 'errno', 'faulthandler', 'gc', 'itertools', 'marshal', 'posix', 'pwd', 'sys', 'time', 'xxsubtype', 'zipimport')

Standard library: sys

- The sys.platform attribute gives information about the operating system
- The sys.version attribute provides information about the interpreter including version, build number, and compiler used. This string is also displayed when the interpreter is started
- >>> sys.platform
  'linux'
  >>> sys.version
  '3.5.0 |Anaconda custom (64-bit)| (default, Oct 19 2015, 21:57:25) \n[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)]'

Standard library: sys

- The sys.stdin, sys.stdout, and sys.stderr attributes hold the file objects corresponding to standard input, standard output, and standard error, respectively
  - Just like every other attribute in the sys module, these may also be changed at any time
- If you want to restore the standard file objects to their original values, use the sys.__stdin__, sys.__stdout__, and sys.__stderr__ attributes
- Output Redirection

```
f = open("datafile.txt", "w")
sys.stdout = f
print "This is going to be written to the file"
sys.stdout = sys.__stdout__
f.close()
```

Standard library: sys

- The sys.exit([status]) function can be used to exit a program gracefully
  - It raises a SystemExit exception which, if not caught, will end the program
  - The optional argument status can be used to indicate a termination status. The value 0 indicates a successful termination, while an error message will print to stderr and return 1

Standard library: os

- The os module provides a common interface for operating system dependent functionality
- Most of the functions are actually implemented by platform-specific modules, but there is no need to explicitly call them as such

Standard library: os

- os.rename(current_name, new_name) renames the file current_name to new_name
- os.remove(filename) deletes an existing file named filename
- There are also a number of directory services provided by the os module
  - os.listdir(dirname) lists all of the files in directory dirname
  - os.getcwd() returns the current directory
  - os.chdir(dirname) will change the current directory

Standard library: os

- >>> os.listdir("/usr")
  ['games', 'include', 'share', 'lib', 'src', 'bin', 'sbin', 'local']
  >>> os.getcwd()
  '/home/hadoop'
  >>> os.rename("test.py", "test2.py")
  >>> os.remove("test2.py")
  >>> os.chdir("/usr")
  >>> os.getcwd()
  '/usr'
  >>> os.listdir(".")
  ['games', 'include', 'share', 'lib', 'src', 'bin', 'sbin', 'local']

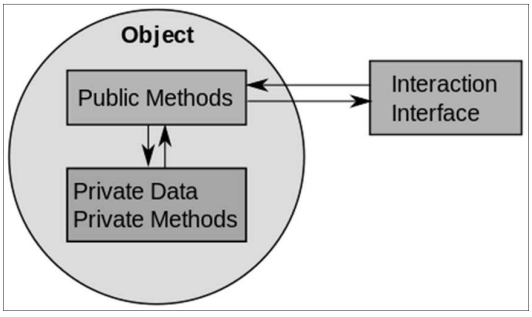
OS Services

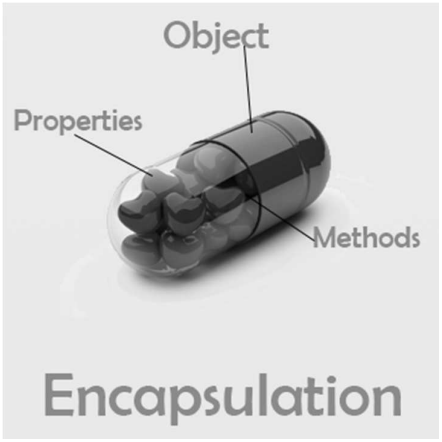
- The os.system(cmd) function executes the argument cmd in a subshell. The return value is the exit status of the command
- >>> os.system("ls")
  0
  >>> os.system("touch newfile.txt")
  0
  >>> os.system("ls")
  newfile.txt
  0

Ch.09 Python Classes

Encapsulation

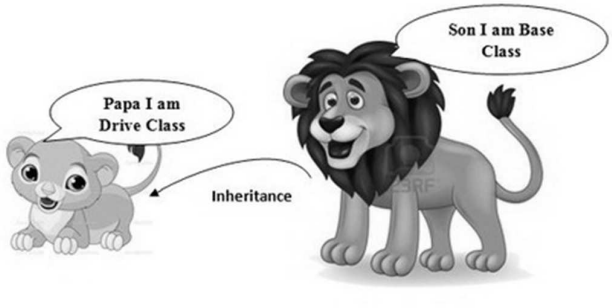
- Used to hide the values or state of a structured data object inside a class, preventing unauthorized parties' direct access to them

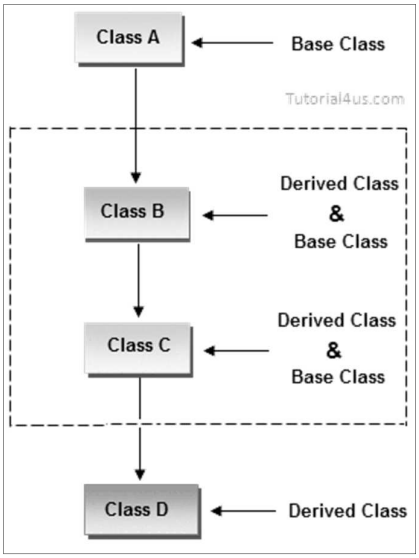




Inheritance

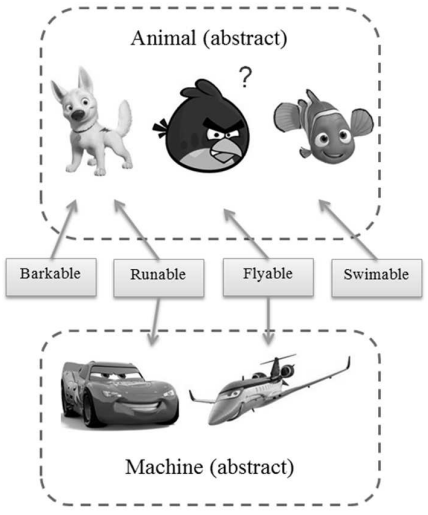
- Define new classes as extensions of existing classes





Polymorphism

- One interface, multiple methods



OOP Terminology Overview

- Class
  - A user-defined prototype for an object that defines a set of attributes that characterize any object of the class
  - The attributes are data members (class variables and instance variables) and methods, accessed via dot notation
- Object
  - A unique instance of a data structure that is defined by its class
  - An object comprises both data members (class variables and instance variables) and methods
- Instance
  - An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle
- Instantiation
  - The creation of an instance of a class

OOP Terminology Overview

- Class variable
  - A variable that is shared by all instances of a class
  - Class variables are defined within a class but outside any of the class's methods
  - Class variables are not used as frequently as instance variables are
- Instance variable
  - A variable that is defined inside a method and belongs only to the current instance of a class
- Data member
  - A class variable or instance variable that holds data associated with a class and its objects
- Method
  - A special kind of function that is defined in a class definition

OOP Terminology Overview

- Function overloading
  - The assignment of more than one behavior to a particular function
  - The operation performed varies by the types of objects or arguments involved
- Operator overloading
  - The assignment of more than one function to a particular operator
- Inheritance
  - The transfer of the characteristics of a class(superclass) to other classes(subclass) that are derived from it
- Method overriding
  - The implementation in the subclass overrides (replaces) the implementation in the superclass by providing a method that has same name

Creating Classes (1)

- The class statement creates a new class definition
- Syntax
  - class ClassName:
    'Optional class documentation string'
    class_suite
    - The class_suite consists of all the component statements defining class members, data attributes and functions

Creating Classes (2)

- Example
  - class Employee:

```
    empCount = 0
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
        Employee.empCount += 1
    def displayCount(self):
        print ("Total Employee {}"
            .format(Employee.empCount))
    def displayEmployee(self):
        print("Name : ", self.name,  ", Salary: ",
            self.salary)
```


Creating Classes (3)

- The variable empCount is a class variable whose value is shared among all the instances of a in this class
  - This can be accessed as Employee.empCount from inside the class or outside the class
- The first method __init__() is a special method, which is called class constructor or initialization method
  - Python will call this method when you create a new instance of this class
- You declare other class methods like normal functions with the exception that the first argument to each method is “self”
  - Python adds the “self” argument to the list for you; you do not need to include it when you call the methods

Using Classes (1)

- Creating Instance Objects
  - To create instances of a class, you call the class using class name and pass in whatever arguments its __init__ method accepts
  - emp1 = Employee("Lisk", 22000)
  - emp2 = Employee("Mary", 55000)
- Accessing Attributes
  - Access the object's attributes using the dot operator with object
  - Class variable would be accessed using class name
  - emp1.displayEmployee()
  - emp2.displayEmployee()

Using Classes (2)

- The hasattr(obj,name) : to check if an attribute exists or not
  - hasattr(emp1, 'salary') # Returns true if 'salary' attribute exists
- The getattr(obj, name[, default]) : to access the attribute of object
  - getattr(emp1, 'salary') # Returns value of 'salary' attribute
- The setattr(obj,name,value) : to set an attribute. If attribute does not exist, then it would be created
  - setattr(emp1, 'salary', 7000) # Set attribute 'salary' at 7000
- The delattr(obj, name) : to delete an attribute
  - delattr(emp1, 'salary') # Delete attribute 'salary'

Static Methods (1)

- class Robot:
  __counter = 0

  def __init__(self):
  type(self).__counter += 1

  def RobotInstances(self):
  return Robot.__counter

  x = Robot()
  print(x.RobotInstances())
  y = Robot()
  print(x.RobotInstances())
- Number of robots has nothing to do with a single robot instance
- Can't inquire the number of robots before we haven't created an instance

Static Methods (2)

- class Robot:
  __counter = 0

  def __init__(self):
  type(self).__counter += 1

  def RobotInstances():
  return Robot.__counter

  print(Robot.RobotInstances()) # OK
  x = Robot()
  y = Robot()
  print(Robot.RobotInstances()) # OK
  print(x.RobotInstances()) # Error
- Possible to access the method via the class name, but we can't call it via an instance

Static Methods (3)

- Add "@staticmethod" decorator directly in front of the method header
  - we can call via the class name or via the instance name without the necessity of passing a reference to an instance to it
- class Robot:

```
    __counter = 0

    def __init__(self):
        type(self).__counter += 1

    @staticmethod
    def RobotInstances():
        return Robot.__counter

print(Robot.RobotInstances())     # OK
x = Robot()
y = Robot()
print(Robot.RobotInstances())     # OK
print(x.RobotInstances())         # OK
```


Data Hiding (1)

- An object's attributes may or may not be visible outside the class definition
- Attributes with a double underscore prefix will not be directly visible to outsiders
  - It's neither possible to read nor write to those attributes, except inside of the class definition itself
  - Python protects those members by internally changing the name to include the class name
  - You can access such attributes as object._className__attrName

Data Hiding (2)

- Example
  - class JustCounter:

```
    __secretCount = 0
    def count(self):
        self.__secretCount += 1
        print (self.__secretCount)

counter = JustCounter()
counter.count()
counter.count()
print (counter.__secretCount)  # Error
print (counter._JustCounter__secretCount)  # Ok
```


Built-In Class Attributes (1)

- Every Python class keeps following built-in attributes and they can be accessed using dot operator like any other attribute
  - __dict__ : Dictionary containing the class's namespace
  - __doc__ : Class documentation string or none, if undefined
  - __name__ : Class name
  - __module__ : Module name in which the class is defined
    - This attribute is "__main__" in interactive mode
  - __bases__ : A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

Built-In Class Attributes (2)

- Example of Employee.__dict__
  - mappingproxy({'__module__': 'test', '__doc__': 'Common base class for all employees', 'empCount': 1, '__init__': <function Employee.__init__ at 0x040188E8>, 'displayCount': <function Employee.displayCount at 0x040188A0>, 'displayEmployee': <function Employee.displayEmployee at 0x04018858>, '__dict__': <attribute '__dict__' of 'Employee' objects>, '__weakref__': <attribute '__weakref__' of 'Employee' objects>})

Destroying Objects (Garbage Collection)

- Python deletes unneeded objects (built-in types or class instances) automatically to free the memory space
- The process by which Python periodically reclaims blocks of memory that no longer are in use is termed as Garbage Collection
- Python's garbage collector runs during program execution and is triggered when an object's reference count reaches zero
- The object's reference count decreases when it is deleted with “del”

Object's Reference Count

- Example
  - a = 40 # Create object <40>
    b = a # Increase ref. count of <40>
    c = [b] # Increase ref. count of <40>

    del a # Decrease ref. count of <40>
    b = 100 # Decrease ref. count of <40>
    c[0] = -1 # Decrease ref. count of <40>

Destructor

- A class can implement the special method __del__(), called a destructor
  - Invoked when the instance is about to be destroyed
  - This method might be used to clean up any non-memory resources used by an instance
- Example
  - class Point:
    def __init__(self, x=0, y=0):
    self.x = x
    self.y = y
    def __del__(self):
    class_name = self.__class__.__name__
    print (class_name, "destroyed")

Class Inheritance (1)

- You can create a class by deriving it from a pre-existing class
  - By listing the parent class in parentheses after the new class name
- The child class inherits the attributes of its parent class, and you can use those attributes as if they were defined in the child class
- A child class can also override data members and methods from the parent
- Syntax
  - class SubClassName (ParentClass1[, ParentClass2, ...]):
    class_suite

Class Inheritance (2)

- Example
  - class Parent: # define parent class
    parentAttr = 100

    def __init__(self):
        print ("Calling parent constructor")
    def parentMethod(self):
        print ('Calling parent method')
    def setAttr(self, attr):
        Parent.parentAttr = attr
    def getAttr(self):
        print ("Parent attribute :", Parent.parentAttr)

Class Inheritance (3)

- class Child(Parent): # define child class
  ```
  def __init__(self):
      Parent.__init__(self)  # parent constructor
      super().__init__()     # parent constructor
      print ("Calling child constructor")
  def childMethod(self):
      print ('Calling child method')
  ```

- ```
  c = Child()              # instance of child
  c.childMethod()      # child calls its method
  c.parentMethod()    # calls parent's method
  c.setAttr(200)    # again call parent's method
  c.getAttr()          # again call parent's method
  ```


Class Inheritance (4)

- Derive a class from multiple parent classes
  - ```
    class A:       # define your class A
    .....
    class B:       # define your calss B
    .....
    class C(A, B):   # subclass of A and B
    .....
    ```
- Use issubclass() or isinstance() functions to check a relationships of two classes and instances
  - The issubclass(sub, sup) returns True, if the given subclass sub is indeed a subclass of the superclass sup
  - The isinstance(obj, Class) returns True, if obj is an instance of class Class or is an instance of a subclass of Class

Overriding Methods

- You can always override your parent class methods for special or different functionality in your subclass
- Example
  - class Parent: # define parent class
    def myMethod(self):
    print ('Calling parent method')
    class Child(Parent): # define child class
    def myMethod(self):
    print ('Calling child method')

    c = Child() # instance of child
    c.myMethod() # child calls overridden method

Overloading Operators

- You could define the __add__ method in your class to perform, for example, vector addition
- Example
  - class Vector:
    def __init__(self, a, b):
    self.a = a
    self.b = b
    def __str__(self):
    return 'Vector (%d, %d)' % (self.a, self.b)
    def __add__(self,other):
    return Vector(self.a + other.a, self.b + other.b)

    v1 = Vector(2,10)
    v2 = Vector(5,-2)
    print (v1 + v2)

Base Overloading Methods

Method	Description & Sample Call
__init__ (self [,args...])	Constructor (with any optional arguments) Sample Call : *obj = className(args)*
__del__(self)	Destructor, deletes an object Sample Call : *del obj*
__repr__(self)	Evaluatable string representation Sample Call : *repr(obj)*
__str__(self)	Printable string representation Sample Call : *str(obj)*
__cmp__ (self, x)	Object comparison Sample Call : *cmp(obj, x)*