THE PYTHON PROGRAMMING LANGUAGE

Part 1: Basics

June 7, 2021

Hemanth P. Sethuram

Email: hemanth.p.sethuram@intel.com

Contents

Contents ...2

Contents ...3

Introduction

What is Python?

- **Python** (`https://www.python.org)is` a widely used, **general purpose programming language** that lets you work quickly and integrate systems more effectively.

- Its design philosophy emphasizes **code readability**, and its syntax allows programmers to express concepts in **fewer lines of code** than would be possible in languages such as C.

What is Python? ...2

- Python supports **multiple** programming paradigms
  - Object Oriented
  - Imperative
  - Functional Programming
  - Aspect Oriented Programming

What is Python? ...3

- It features a **dynamic type system** and **automatic memory management**.
- It comes with a large standard library and hence the term **Batteries Included!**.

History

- **Python** was conceived and implemented in 1989 by **Guido von Rossum**, popularly called in the Python community as **BDFL - Benevolent Dictator For Life**.

- The initial language already had **late binding** and a combination of reference counting and cycle-detecting **garbage collector** for automatic memory management.

- **Python 2.0** was release in October 2000 with a full fledged **garbage collector**, support for **Unicode**, and a well defined development process. New features are now added through **PEPs - Python Enhancement Proposal**.

- **Python 3.0**, a backwards-incompatible revision, was released in 2008. A well documented migration path is provided to Python 3.0 from earlier versions. **Python 2.7.x**, the last in the Python 2.x series will be maintained and supported till the year 2020.

Industry Usage

Python is now widely used across a spectrum of application domains.

- in scripting as a general purpose glue language, with good support in standard library to access OS facilities, string manipulation, regular expressions, etc.
- in server side Web applications development (e.g., Django, flask, twisted, etc.)
- in financial services applications

Industry Usage ...2

- in scientific applications (e.g., Scipy, Numpy, matplotlib) and simulations
- in statistics and machine learning (e.g., scikit-learn, pandas)
- in general purpose engineering tools (viz., testing, code management, etc.)

Because, of its easy interfacing with external libraries, it is used as an application development language with compute-intensive jobs wrapped in C/C++ libraries.

Dipping Our Feet In...

Running a Python Program

- Store your program in a `.py` file and run from command-line

```
python myfile.py
```

- Typing `python` on the command-line opens an interactive interpreter. You can type inside the interpreter session and execute the statements.
- Or, use the `IDLE` application that comes default with your **Python** installation. It allows to enter statements in the interpreter or a file editor.

How does a Python program look like?

```
import os,sys
def count(filename):
    '''function: count(filename)
    Reads the contents of the file given in 'filename' and
    counts lines, words and characters in that file.
    Returns (lineCount, wordCount, charCount).
    '''
    lineCount, wordCount, charCount = 0, 0, 0
    f = open(filename, "r")
    for line in f: # iterate for every line
        lineCount += 1
        words = line.split() # split line into words
        wordCount += len(words)
        for word in words: # count chars in each word
            charCount += len(word)
    return (lineCount, wordCount, charCount)
```

```
if __name__ == "__main__":
    if len(sys.argv) < 2:
        print("Error: Insufficient arguments")
        print("Usage:", sys.argv[0], "<filename>")
        sys.exit(1)

    filename = sys.argv[1]
    if os.path.isfile(filename):
        lc, wc, cc = count(filename)
        print("File", filename, "contains",
                lc, "lines,", wc, "words and",
                cc, "characters")
    else:
        print(filename, "is not a regular file")
        sys.exit(1)
```

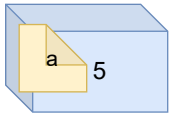
A Simple Python Program

```
principle = 1000        # Initial amount
rate = 0.05             # Interest rate
numyears = 5            # Number of years
year = 1
while year <= numyears:
    principle = principle * (1 + rate)
    print(year, principle)
    year += 1
del rate # example to show unbinding of a name
print(rate) # raises an exception as rate not bound
```

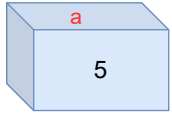
Python

C/C++/Java

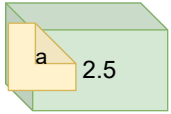
a = 5



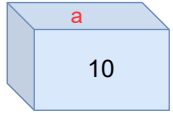
int a;
a = 5;



a = 2.5

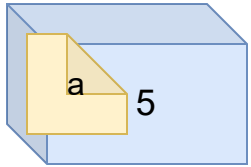


a = 10

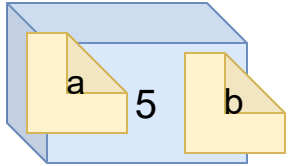


del a

a = 5

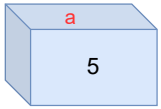


b = a

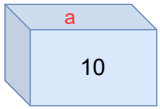


C/C++/Java

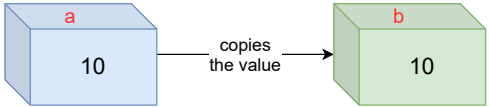
int a;
a = 5;



a = 10



int b;
b = a;



- Names (`principle`, `rate`, etc.) are assigned (bounded) to the objects. No type declaration is required. The same name can be bound to different types at different times.

- Statements are terminated by newline — no semicolons.

- Indentation determines block structure — no braces. Always use 4 spaces for indenting; do not use tabs.

- Characters following a `#`, until the end of a line, are comments.

Simple Operations

Arithmetic Operations

- Python comes with the usual set of arithmetic operators on numbers. (`+`, `-`, `*`, `/`, `%`, `//`).
- `/` peforms floating point division, `//` gives quotient and `%` gives the remainder.
- Exponentiation is supported through `**` operator.

Truth and Logical Operations

- `True` and `False` are the two Boolean values.
- Logical operators supported: `and`, `or` and `not`.
- All objects have an inherent boolean true or false value.
- Any nonzero number or nonempty object is true.
- Zero numbers, empty objects, and the special object `None` are considered false.

Truth and Logical Operations ...2

- Comparisons and equality tests are applied recursively to data structures.
- Comparisons and equality tests return `True` or `False`.
- Boolean `and` and `or` operators return a true or false operand object.
- Boolean operators stop evaluating (“short circuit”) as soon as a result is known.

Comparison

- All regular comparison operations are supported (e.g., `>`, `>=`, `<`, `<=`, `!=`, `==`).
- `=` is used for assignment and `==` is used to check equality of values. Unlike the C language, assignment is not allowed inside a conditional statement.
- `if x is y:`, checks if `x` and `y` refer to the **same object**, while `x == y` checks if `x` and `y` have the **same values**.

Bitwise Operations

- Python also supports bitwise operations on integer types.
  - bitwise or (|)
  - bitwise and (&)
  - exclusive or (^)

```
>> 3 & 5
1
>> 3 | 5
7
>> 3 ^ 5
6
```

– You can get the binary, octal and hexadecimal string representation of a number by using the functions `bin()`, `oct()` and `hex()`, respectively.

```
>> bin(35)
0b100011
>> oct(35)
0o43
>> hex(35)
0x23
```

Conditionals: if, elif, else

- `if` statement can be used to check for truthfulness of an expression.
- An optional `else` block can be used to execute statements if the condition is false.
- Multiple alternatives can be evaluated using `elif` for each subsequent alternative. Unlike the C language, there is no `switch` statement in Python.

```
if suffix == ".htm":
    content = "text/html"
elif suffix == ".jpg":
    content = "image/jpeg"
elif suffix == ".png":
    content = "image/png"
else:
    content = "Unknown content type"
```

Conditionals: while

- `while` checks for a condition and executes the following block as long as the condition is true.
- `break` jumps out of the enclosing group. `continue` jumps to the top of the enclosing loop. `pass` is an empty statement placeholder.
- It also has an `else` block, which is executed if the `while` block was not exited through a `break` statement.

```python
s = "Hello World"
i = 0
while s[i]:
    print(s[i])
    if s[i] == ' ':
        break
    i += 1
else: # executed only if break was not executed in while block
    print("No spaces found in string")
```

Functions

Function Definitions

- A function is defined using the `def` keyword. Function invocation is similar to that of the C language.
- Unlike **C**, functions can return multiple values.
- If no `return` statement is present, it is treated as returning `None`.

```python
def divide(a, b):
    '''This module divides 2 numbers
       and returns a quotient and a reminder.
    '''  # This is displayed as function documentation
    if not b:
        print("Error: Denominator cannot be zero!")
        return None
    q = a // b # integer division
    r = a - q*b
    return (q,r)

# Note: functions can return multiple values
quotient, remainder = divide(1456, 33)
```

Function Arguments

- Function arguments can have default values. While invoking, if a required argument is not given, its default value will be used, if defined.
- When calling a function, arguments can be specified by position or by explicitly by assigning a value to their names.

```
def add(x, y, z=0, w=0):
    return x+y+z+w

add(2, 3, 4, 5) # all arguments specified
add(5, 7, 9) # only 3 arguments specified. Default value used for 'w'
add(5, w=3, z=10, y=4) # key=value in any order after positional args
add(z=10, y=13, 2, 3) # illegal. key=value should be in the end
```

lambda

- Unnamed functions are often used as arguments to other functions. E.g.,
  - as ordering criteria for sort functions
  - to indicate operations to be applied on each element of a sequence, etc.
- The `lambda` keyword allows us to define anonymous functions.

```
def greater(a, b, islarger_func):
    if(islarger_func(a, b)):
        return a
    else:
        return b

big = greater(3, 5, lambda x, y: x > y)
print("The greater number of 3 and 5 is", big)
```

Handy Built-in Functions

- `dir(pyobj)` lists the methods supported by a Python object.
- `type(pyobj)` returns the type of the given object
- `id(pyobj)` returns the unique id of an object.
- `help(pyobj)` prints the documentation of the Python object
- `len(pyseq)` returns the length of a collection like strings, lists, dictionaries, sets, etc.

```
>> s = "Hello"
>> t = "World"
>> x = 3
>> dir(s)
>> type(s), type(t)
>> id(s), id(t), id(x)
>> id(s+t) # check if it is different from s and t
>> help(s.replace)
>> dir(__builtins__)
>> len(s) # length of the string
```

Modules

Module Definition

- Python programs are organized as modules and packages
- Any Python source file can be used as a module.
- Any directory with an `__init__.py` can be used as a module.
- Every module has its own namespace and all the global names in the module resides in this namespace. This avoids name clashes.

Using Modules with import

- Python provides the `import` statement with which modules can be used in our own programs.
- There are multiple forms of this `import` statement that can be used. Some of the commonly used forms are listed here.
- Modules are searched in the following paths on your filesystem.
  - current directory
  - directories specified in `PYTHONPATH` environment variable
  - standard library directories (usually `C:/Python37/lib/site-packages`)
  - list of directories specified in a `.pth` file in your path.

import Examples

```
>> import os, sys
>> os.listdir('.') # use module.name to access name
>> from math import sin, cos, pi # get specific objects into current namespace
>> sin(pi/4) # objects can be now directly used
>> from decimal import Decimal as D # get an object and rename
>> D('3.45') / D('2.87') # use the new name
```

A note on module exports

If you have the following in source file `example.py`.

```
# file: example.py
def foo(): # this function is available for import
    pass

def bar(): # this function is available for import
    pass

N = 2.5  # this value is available for import

_D =  4 / 5   # this value is available for import

if __name__ == "__main__": # only available when run directly
    M = 3
    print("Running the file directly. M =", M)
    print("Outside module import")
```

What names are imported?

- If you run this file directly as `python example.py`, all lines are executed.
- If you import this file as a module, then the block inside the `if __name__ ...` statement is not executed. This is useful for testing a module.

```
>> import example
>> dir(example) # only foo, bar and N are visible, not M
```

Core Data Types

Built-in Objects Overview

- Python comes with a set of built-in objects containing many high-level data structures.
- These objects provide comprehensive and powerful methods making it easy to perform powerful operations on these datastructures.

Built-in Objects Overview ...2

Object Type	Example literals/Creation
Numbers	`1234, 3.1415, 3+4j, 0b111,`
	`Decimal(), Fraction()`
Strings	`'spam', "Bob's", b'\xa0\x1c',`
	`u'\x68\x65\x72\x6F'`
Lists	`[1, [2, 'three'], 4.5], list(range(10))`
Dictionaries	`{'food': 'spam', 'taste': 'yum'},`
	`dict(hours=10)`

Built-in Objects Overview ...3

Object Type	Example literals/Creation
Tuples	`(1, 'spam', 4, 'U'), tuple('spam'),`
Files	`open('eggs.txt'), open(r'C:/ham.bin', 'wb')`
Sets	`set('abc'), {'a', 'b', 'c'}`
Other core types	Booleans, types, None
Program unit types	Functions, modules, classes

Numbers

- **Python** supports *integers*, *floating point numbers* and *complex numbers* as built-in types.
- Unlimited precision decimals are supported through the `decimal` module.
- Fractions are supported through `fractions` module.

```
>> 3 + 2
>> 5 / 7
>> (3 + 4j) / (4 - 5j)
>> from decimal import Decimal
>> Decimal('3.49') + Decimal('4.23')
>> from fractions import Fraction
>> Fraction(3,5) + Fraction(5,3)
```

Sequence Types

- **Sequences** represent ordered sets of objects indexed by *integers*.
- These include *strings*, *lists* and *tuples*. Strings and tuples are **immutable** while lists are **mutable**.
- Lists allow *insertion*, *deletion* and *substitution* of elements.
- Indexing by *negative numbers* give elements from the *end*.

Operations supported by all sequences

Operation	Description
`s[i]`	Returns element `i` of a sequence
`s[i:j]`	Returns a slice (items at indices `i` to `j-1`)
`s[i:j:stride]`	Returns a slice which are `stride` items apart
`s[:j]`	Returns a slice from the begining of
	`s` till index `j-1`
`s[j:]`	Returns a slice starting at index `j`
	till the end of `s`

Operations supported by all sequences ..2

Operation	Description
`len(s)`	Number of elements in `s`
`min(s)`	Minimum value in `s`.
	Elements should be orderable.
`max(s)`	Maximum value in `s`.
	Elements should be orderable.
`sum(s [,initial])`	Sum of items in `s`.
	Items should be numbers.

Operations supported by all sequences ..3

Operation	Description
`all(s)`	Checks whether all items in `s` are `True`.
`any(s)`	Checks if at least one item in `s` is `True`.

Operations supported by Mutable Sequences

Operation	Description
`s[i] = v`	Item assignment
`s[i:j] = t,u,v`	Slice assignment
`s[i:j:stride] = [t,u,v,x]`	Extended slice assignment
`del s[i]`	Item deletion
`del s[i:j]`	Slice deletion
`del s[i:j:stride]`	Extended slice deletion

Strings

- **Strings** are *immutable* sequence of unicode code points.
- There are 3 ways of specifying a string literal:
  - with single quotes: `'a string with "embedded" double quotes'`
  - with double quotes: `"a string with 'embedded' single quotes"`
  - with triple quotes: `'''this can be`

    `multi-line'''`

Strings ..2

- Some characters preceded by a backslash are interpreted with special meanings, similar to C language.
- Strings prefixed with the character `r` are not interpreted for special meanings. E.g., in `r"Hello \n World"`, `\n` is not interpreted.
- Unicode strings can be prefixed by `u` and bytestrings can be prefixed by `b`.
- Python strings come with a rich set of methods.

Strings Examples

```
>> s = "Hello"
>> t = "World"
>> s + t
>> dir(s)
>> s.replace("e","a")
>> s[3] # get 4th element
>> s[2:4] # get elements 3 to 4 excluding last index
>> t[-3] # get 3rd element from the end
>> s[-3:-1]
>> s[3] = "a" # changing a string not allowed. Should give TypeError
```

Strings Interpolation

- There are many ways in which you can create new strings from a template string with placeholders by supplying the values for the placeholders.

```
>>> name = "Harry"; age = 32
>>> f"{name} is {age} years old"
'Harry is 32 years old'
>>> "{0} is {1} years old".format("Harry", 32)
'Harry is 32 years old'
>>> from math import pi, sin
>>> "pi={0:0.4f} sin(pi/4)={1:0.2f}".format(pi, sin(pi/4))
'pi=3.1416 sin(pi/4)=0.71'
>>> "{name} is {age} years old".format(age=32, name="Harry")
'Harry is 32 years old'
```

```
>>> for C in range(0,30,10):
  F = (9*C/5) + 32
  print("Celsius:{C} Fahrenheit:{F}".format(C=C, F=F))
Celsius:0 Fahrenheit:32.0
Celsius:10 Fahrenheit:50.0
Celsius:20 Fahrenheit:68.0
```

Lists

- **Lists** are mutable ordered sequences. They can contain any Python objects including Numbers, Strings and even other lists.
- They are indexed by integers and provide fast random access to any element.
- Lists can be grown in *either* direction. They can be modified in-place by assigning an indexed location to a new Python object.
- Only object references are stored and hence quite efficient.
- In total, they provide a very powerful data structure which is of variable length, containing heterogenous objects and can be arbitrarily nested.

Lists examples

```
>> x = [] # an empty list
>> y = list() #through the list() function
>> x.append(3) # grow the list from the end
>> x.append(5)
>> x.extend(["Hello", "World"]) # get elements from another list and grow
>> 5 in x # check if given object is present in list
True
>> z = x + [5, 6] # create a new list by concatenation, x is unchanged
[3, 5, "Hello", "World", 5, 6]
>> x.pop() # remove and return the last item appended
World
>> x.insert(0, "Good") # insert an object at the beginning
>> x[2] = 25 # point a location to a different object
>> dir(x) # explore other methods of lists
```

A Detour to 'for loops'

– Iterating over elements of a sequence can be done using a `for` loop construct.

```
for item in sequence: #returns each element in the sequence
    if item == some_value:
        break  # stop iteration and jump out
    if item == other_value:
        continue # skip to the next iteration
    process(item) # process item in this iteration
else: # an optional else block
    # this block is entered if break was not hit above
```

Example 'for' loop

Put the following in a file `example_for.py` and run.

```
1  import sys
2  for arg in sys.argv: #sys.argv contains a list of command line arguments given
3      print(arg)
4  x = [3, 5, "Hello", "World"]
5  for item in x:
6      if type(item) is str:
7          continue   #skip processing the strings
8      print(item)
9  else:
10     print("Did not break")
```

Exercise on Lists

Write a program `textstats.py` as follows.

- Should be able to take 1 or more command line arguments specifying text files.
- Should count the following items: number of lines, words and characters in a given text file.
- Should print the file name with the above 3 counts.

Tuples

Tuples are sequences containing Python objects similar to lists except that they are **immutable**.

- **Ordered collection** of arbitrary objects
- Accessed by offsets, similar to lists
- **Immutable sequence**
- Fixed-length, heterogeneous and arbitrarily nestable
- Arrays of object references

Tuples Examples

```
# created using parentheses
t = (2, 4, "Harry", "Sally")
# alternatively through tuple()
t = tuple([2, 4, "Harry", "Sally"])
type(t)
print(t[1]) # subscript access
t[1] = 7 # should throw exception, no mutation
t = (35) # assign an integer to name 't'
t = (35,) # assign a tuple containing an integer.
def func():
    # function can return multiple values through tuple
    return 3,5
a, b = func() # get multiple values
t = func() # get the tuple
print(type(t))
```

Files

- Similar to the C language, Python supports operations on file and file-like objects.
- `open(filename, mode)` opens a file in read or write mode and returns a file handle.
- `read()` and `write()` calls can be used to read/write bytes into the file.
- By default, all reads return a string unless the file is opened in “b” mode.

Files ...2

- `readline()` and `readlines()` allow you to read the file line by line. Similarly, `writelines(stringList)` writes each string in the list to a separate line.
- `sys.stdin`, `sys.stdout` and `sys.stderr` allows access to the console I/O.
- You can use `flush()` and `close()` calls after the file operations are done.

Iterating over lines in a file

There is a better way to handle each line in a file using the `for` loop.

Enter the following in a file `example_cat.py` and run. You should get output similar to the unix `cat` program.

```
1  import sys
2  for filename in sys.argv[1:]: # iterate over every file given in command line
3      f = open(filename, "r")
4      for line in f: # open the current program file in read mode
5          print(line) # print the current line
6      f.close() # close the file
```

Exercise on Searching

Write a program `textsearch.py`, similar to the unix program `grep`.

- Take at least 2 arguments from command line; the first argument should be the search string and rest of the arguments are name of text files.
- For each file, search every line for the existing of the specified substring. If present, print the line preceded by file name.
- Make use of the `find()` method on each line string. Study documentation of `find()` and be careful on its return value for success and failure.

Dictionaries

Dictionary is a powerful built-in data structure available in Python. It can replace many of the searching algorithms and data structures from low level languages.

- **Unordered** collection of objects accessed by **immutable** keys.
- Accessed by key, **not** offset position
- Variable-length, heterogeneous, and arbitrarily nestable.
- **Mutable mappings**. Contents can be changed *in-place*.
- Objects are stored by their references; no copies involved.

Dictionaries example

```
d = {"score": 90, "greeting": "Hello", 3: 5} # {} literal
# Alternatively
d = dict(score=90, greeting="Hello")
# Or with a list of (key,value) pairs
d = dict([('score',90), ('greeting','Hello'), (3,5)])
d["greeting"] = "Hi"  # modify an item
scoreVal = d["score"] # access through subscripting
for key in d: # iterating gives keys
    print("Key:", key)
for (key,val) in d.items(): # use items() method to get both
    print("Key:", key, "Value:", val)
m = [7, 9, "Greet", "Sky"]
e = {"name":"Bill", "age": 32}
d[25] = m # can contain other Python objects
d["nest"] = e # including other dictionaries
```

Exercise on Dictionaries and Lists

Write a program `top5.py` such that the program,

- takes a text file name as a command line argument
- collects all the words in the text file
- alphabetically sorts these words
- prints the first 5 words in the alphabetical order

Tips

- Use `sort()` or `sorted()` functions to sort a list.
- Use dictionary to store each word.
- Use `keys()` method on the dictionary to get a list of keys.

Another Exercise on Dictionaries and Lists

Write a program `wordhist.py` such that the program,

- takes a text file name as a command line argument
- counts the number of times each word appears in the file
- prints the top 5 frequently occuring words

Tips

- Make each unique word as a key in a dictionary.
- Every time you encounter the same word, increment the count in the dictionary entry keyed by the word.
- Make a reverse dictionary by using values as keys from the previous dictionary.
- Sort the `keys()` list from the second dictionary and print the top 5 entries from the second dictionary using the sorted keys.

Sets

- A **set** is an unordered collection of unique items.
- Unlike sequences, sets provide **no indexing or slicing** operations.
- Unlike dictionaries, **no key values** are associated with the objects.
- Like dictionary keys, the objects placed into a set must be /immutable/.
- There are 2 kinds of sets. `set` is a mutable set and `frozenset` is an immutable set.
- Some methods of set types are - `difference()`, `intersection()`, `union()`, `isdisjoint()`, `issubset()`, `issuperset()`.

Sets Example

```
# note difference with dict: no values
>> s = {1, 5, 10, "Hello", "World"}
>> s = set([1, 5, 10, "Hello", "World"]) # alternative definition
>> t = set([3, 2, 1, "World"])
>> s.difference(t) # elements in s but not in t
{5, 10,"Hello"}
>> s.intersection(t) # elements in both s and t
{1, "World"}
>> s.union(t) # combine elements of s and t
# elements either in s or t but not in both
{1, 5, 10, "Hello", "World", 3, 2}
>> s.symmetric_difference(t)
{2, 3, 5, "Hello", 10}
>> u = set([1, "Hello"])
>> u.issubset(s)
True
>> s.issuperset(u)
True
```

Exceptions

Exceptions Basics

- **Exceptions** are events that can modify the flow of control through a program.
- They can be triggered manually using the `raise` statement.
- They can be triggered conditionally using the `assert` statement.

Exceptions Basics ...2

```
f = open('foo', 'r')
try: # fence the code block that could be troublesome
    data = f.read()
    result = process(data) # can potentially cause exception
except IOError as e:
    error_log.write('Unable to open foo : %s\n' % e)
else:
    print(result) # print if everything went well
finally: # always executes this block
    f.close()
```

Handling Multiple Exceptions

Example handling multiple exceptions separately.

```
try:
    # do something
except IOError as e:
    # Handle I/O error
except TypeError as e:
    # Handle Type error
except NameError as e:
    # Handle Name error
```

Handling Multiple Exceptions ...2

You can club together handling of multiple exceptions.

```
try:
    # do something
except (IOError, TypeError, NameError) as e:
    # Handle I/O, Type, or Name errors
```

Uses of Exceptions

```
It is easier to ask for forgiveness than for permisssion
```

- Error Handling - can separate error handling code from main control flow. Keeps the code clean and helps debugging.
- Event Notification - can be used to signal valid conditions (like a search failure) without resorting to flags.
- Special case handling - rarely occuring error conditions can be handled at the top of your code without peppering condition checks throughout.
- Unusual control flows - can be used as a structured `goto` to jump from arbitrary nested code with safe stack unwinding and object cleanup

Classes

Introduction to Classes

- A **class** defines a set of attributes that are associated with, and shared by, a collection of objects known as **instances**.
- A class is most commonly a collection of functions (known as **methods**), variables (which are known as **class variables**), and computed attributes (which are known as **properties**).
- All class and instance variables are *public by default*. By convention, names starting with underscores are deemed for internal use by the class author.

Class Definition

```
1  class InsufficientBalanceError(Exception):
2      pass # defining our own exception. Derived from Exception
3
4  class Account():
5      num_accounts = 0 # class variable; executed during definition
6      def __init__(self, name, initialBal=0):
7          self.name = name
8          self.balance = initialBal
9          Account.num_accounts += 1
10     def __del__(self):
11         Account.num_accounts -= 1
12     def deposit(self, amount):
13         self.balance += amount
14     def withdraw(self, amount):
15         if amount < self.balance:
16             self.balance -= amount
17         else:
18             raise InsufficientBalanceError
```

Class Definition continued..

- The instance method is a function that operates on an instance of the class. This instance is *automatically passed as the first argument* when the method is invoked.
- The instance argument is conventionally called, `self`.
- Class variables are shared among all instance of a class.

Instantiation

```
h = Account("Harry", 50000) # instantiate an Account class
s = Account("Sally", 75000) # instantiate another class
print(dir(h)) # examine the contents
h.deposit(23000) # calls instance method
print(h.balance) # access instance variable
h.withdraw(12300)
print(h.balance)
print(Account.num_accounts) # access class variable
del h # delete the instance. __del__ automatically called
print(Account.num_accounts)
print(Account.__dict__) # attributes of Accound class
print(s.__dict__) # attributes of instance s
```

Instantiation continued..

- Object instantiation is similar to calling a function with arguments.
- `__init__()` method is automatically called after the instance is created.
- All object methods are automatically called with the object as the first parameter.
- Destroying an object will automatically call the `__del__()` method.

Inheritance

- Inheritance is a mechanism for creating a new class that **specializes** or **modifies** the behavior of an existing class

```
class SpecialAccount(Account): # Account is the base class
    def __init__(self, name, initialBal=0):
        super().__init__(name, initialBal) # super() calls parent
    def withdraw(self, amount):
        if super().balance < 25000:
            super().withdraw(amount + 500) # add penalty
        else:
            super().withdraw(amount)
```

- If the search for an attribute doesn't find a match in the instance or the instance's class, the search moves on to the base class.

Multiple Inheritance Example

```
class FundsTransfer():
    fee = 5.0
    def transfer_fee(self):
        return FundsTransfer.fee
class FundsReceive():
    fee = 2.5
    def receive_fee(self):
        return FundsReceive.fee
class TransferAccount(SpecialAccount,FundsTransfer,FundsReceive):
    def deposit(self, amount):
        total = amount - self.receive_fee()
        super(TransferAccount, self).deposit(total) #call parent
    def withdraw(self, amount):
        total = amount + self.transfer_fee()
        super(TransferAccount, self).withdraw(total)

print(TransferAccount.__bases__) # prints the immmediate parents
print(TransferAccount.mro()) # prints method resolution order
```

Multiple Inheritance

- Multiple inheritance allows Python classes to inherit and specialize behavior from multiple classes.
- *Multiple base classes* are specified as *arguments* to class definition statement
- When a name is accessed on an object instance which has multiple base classes, the name is searched from the current object instance through its parent class hierarchy until a match is found.
- The name lookup order is called *method resolution order* and can be found in the class method `mro()`. First an instance method having the given name is searched, then a `classmethod` and finally a `staticmethod` is searched. If no such name exists in any of the above categories, an `AttributeError` exception is raised.

```
>>> print(TransferAccount.mro())
(<class '__main__.transferaccount'="">,
 <class '__main__.specialaccount'="">, <class '__main__.account'="">,
 <class '__main__.fundstransfer'="">,
 <class '__main__.fundsreceive'="">, <class 'object'="">)
```

Operator Overloading

- Many builtin Python operators can be made to work with user defined class instances by re-defining some built-in methods.
- For example, `__add__()` and `__sub__()` methods can be defined in your own class. This will allow the symbols **+** and **-** to be used on your class instances.

Operator Overloading Example

```
class MyList():
    def __init__(self):
        self.x = [] # an empty list to start with
    def __add__(self, other):
        if type(other) in [int, float, str]:
            self.x.append(other) # append the element
        elif type(other) is list:
            self.x.extend(other) # i.e. add 2 lists
        else:
            raise ValueError
    def __str__(self): # define how this object has to be printed
        return "Have a Good Day"
```

```
>> m = MyList()
>> m + 3
>> m + "Hello"
>> m + 4.5
>> m + [4, 8, "World"]
>> print(m.x)
[3, "Hello", 4.5, 4, 8, "World"]
>> print(m) # calls __str__() method on m
'Have a Good Day'
```

END

Visit https://www.python.org