GAMAKA
Artificial Intelligence Solutions

Python

Compiled by Mahesh Kaneri & Pranav Chandaliya

1. COMMON BUILT-IN DATA TYPES IN PYTHON:

There are several built-in data types in Python. Python provides type() and isinstance() functions to check the type of these variables. These data types can be grouped into the following categories-

None – None keyword represents the null keyword in Python.

Numeric – There are three distinct numeric types – integers, floating point numbers, complex numbers. In addition Booleans are subtype of integers.

- int – Stores integer literals including hex, octal and binary numbers as integers.
- float - Stores literals containing decimal values and/or exponent sign as floating-point numbers.
- complex – Stores complex numbers in the form (A + Bj) and has attributes: real and imag.
- bool – Stores boolean value (True or False).

Sequence – According to Python there are four basic Sequence Types – lists, tuples, and range.

- list – Lists are mutable sequences, typically used to store collections of homogeneous items.
- tuples – Tuples are immutable sequences, typically used to store collections of heterogeneous data.
- range – The range type represents an immutable sequence of numbers and is commonly used for looping a specific number of times in for loops.
- string – Immutable sequence of characters to store textual data.

Mapping – Mapping objects are mutable and there is currently only one standard mapping type, the dictionary.

- dict – Mutable unordered collection of items.

Set – Python has two built-in set types – set and frozenset.

- set – Mutable unordered collection of distinct hashable objects.
- frozenset – Immutable collection of distinct hashable objects.

2. CONTROL FLOW TOOLS IN PYTHON:

2.1 if statements

- The if statement is used in Python for decision making.
- It contains a body of code which runs only when the condition given in the if statement is TRUE.
- If the condition is FALSE, then the optional else statement runs which contains some code for the else condition.
- The keyword 'elif' is short for 'else if' and is useful to avoid excessive indentation.

2.2 for Statements

- It has an ability to iterate over the items of any sequence, such as a list or a string, in order that they appear in the sequence.
- The for statement in Python differs a bit from what you may be used to in C or Pascal.

2.3 The range () Function

- If you do need to iterate over a sequence of numbers, the built-in function range () comes in handy. It generates arithmetic progressions.
- The given end point is never part of the generated sequence.
- **Example:** range (5,10)
  Output: 5,6,7,8,9

2.4 Break and Continue statements

Break Statement –

- The Break statement, like in C, breaks out of the innermost enclosing for or while loop.
- The Break statement terminates the loop immediately, and control flows to the statement after the body of the loop.

Continue Statement –

- The Continue statement, also borrowed from C, continues with the next iteration of the loop.
- The Continue statement terminates the current iteration of the statement, skips the rest of the code in the current iteration and control flows to the next iteration of the loop.

2.5 Pass Statements

- The Pass statement does nothing. It can be used when the statement is required syntactically but the program requires no action.
- Pass keyword in Python is generally used to fill-up empty blocks and is similar to an empty statement represented by semi-colon in languages such as Java, C++.

2.6 Defining Functions

The keyword def introduces a function definition. It must be followed by the function name and the parenthesized list of formal parameters.

- The statements that form the body of the function start at the next line, and must be intended.

Example:
```
def fib(n):    #write Fibonacci series upto n
    a , b = 0 , 1
    while a < n:
        print(a, end=' ')
        a , b = b , a+b
    print()
    #Now call the function we just defined
    fib(2000)
```
Output: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597

2.7 More on Defining Functions

2.7.1 Default Argument Values

- The most useful form is to specify a default value for one or more arguments.
- This creates a function that can be called with fewer arguments than it is defined to allow.
- **Example:**
```
def greet(name, msg='Good morning!'):
    print("Hello", name + ' , ' + msg)
greet ("Kate")
greet("Bruce", "How do you do!")
```
Output:
```
Hello Kate, Good morning!
Hello Bruce, How do you do?
```

2.7.2 Keyword Arguments

- Functions can also be called using keyword arguments of the form kwarg = value.
- **Example:**
```
def SomeFun (**kwargs):
    for argumentname, argumentvalue in kwargs.items()
        print(argumentname, argumentvalue)
```

SomeFun(name = 'Kiran' ,age = 23, role = 'Engineer')

Output: name = Kiran
age = 23
role = Engineer

2.7.3 Special Parameters

- By default, arguments may be passed to a Python function either by position or explicitly by keyword.
- For readability and performance, it makes sense to restrict the way arguments can be passed so that a developer need only look at the function definition tp determine if items are passed by position, by position or keyword, or by keyword.
  - **Positional – or – Keyword Arguments**
    If / and * are not present in the function definition, arguments may be passed to a function by position or by keyword.
  - **Positional – Only Parameters**
    It is possible to mark certain parameters as positional – only. If positional – only, the parameters order matters, and the parameters cannot be passed by keyword.

    Positional – only parameters are placed before a / (forward slash).
  - **Keyword – Only Arguments**
    To mark parameters as keyword – only, indicating the parameters must be passed by keyword arguments, place an * in the arguments list just before the first keyword – only parameters.

Example:
```
def standard_arg (arg):
    print(arg)
def pos_only_arg (arg, /):
    print(arg)
def kwd_only_arg (*, arg)
    print(arg)
```

2.7.4 Arbitrary Argument Lists

- The least frequency used open is to specify that a function can be called with an arbitrary number of arguments.
- These arguments will be wrapped up in a tuple. Before the variable number of arguments, zero or more normal arguments may occur.
- **Example:** def concat (*args, sep = " / "):
  return sep.join(args)
  concat("earth" , "mars" , "venus")
  Output: 'earth/mars/venus'

2.7.5 Unpacking Argument Lists

- The reverse situation occurs when the arguments are already in a list or tuple but need to be unpacked for a function call requiring separate positional arguments.
- For instance, the built – in range() function expects separate start and stop arguments.
- If they are not available separately, write the function call with the * operator to unpack the arguments out of the list or tuple.
- **Example:** list(range(3,6)) #normal call with separate arguments
  Output: [3,4,5]

  args = [3,6]
  list(range(*args) #call with arguments unpacked from a list
  Output: [3,4,5]

2.7.6 Lambda Expressions

- Small anonymous functions can be created with the lambda keyword.
- This function returns the sum of its two arguments lambda a , b : a + b.
- Lambda functions can be used wherever function objects are required.
- **Example:** list = [3,2,7,5,8,9,11,1]
  result = filter(lambda N:N>5,list)
  print(list(result)
  Output: [7,8,9,11]

2.7.7 Documentation Strings

- A documentation string is a string iteral that occurs as the first statement in a module, function, class, or method definition.
- They are used to document our code.

- Such a docstring becomes the _doc_ special attribute of that object.

2.7.8 Function Annotations

- Function annotations are completely optional metadata information about the types used by user – defined functions.
- Annotations are stored in the _annotations_ attribute of the function as a dictionary and have no effect on any other part of the function.

3. DATA STRUCTURES:

3.1 More on Lists

The list data types has some more methods:

list.append (x) –

- Add an item to the end of the list. Equivalent to a[len(a):] = [x].

list.extend (iterbale) –

- Extend the list by appending all the items from the iterable. Equivalent to a[len(a):] = iterable.

list.insert (i,x) –

- Insert an item at a given position.
- The first argument is the index of the element before which to insert, so a.insert(0,x) inserts at the front of the list, and a insert(len(a),x) is equivalent to a.append(x).

list.pop ([i]) –

- Remove the item at the given position in the list, and return it.
- If no index is specified, a.pop() removes and returns the last item in the list.

list.clear () –

- Remove all items from the list. Equivalent to del a[:].

list.index(x[, start[, end]]) –

- Return zero – based index in the list of the first item whose value is equal to x. Raises a Value – Error if there is no such item.
- The optional arguments start and end are interpreted as in the slice notation and are used to limit the search to a particular subsequence of the list.

list.count (x) –

- Return the number of times x appears in the list.

list.sort (*,key=None, reverse=False)

- Sort the items of the list in place.

list.reverse () –

- Reverse the elements of the list in place.

list.copy () –

- Return a shallow copy of the list. Equivalent to a[:].

3.1.1 Using Lists as Stacks

- The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved **("last-in, first-out")**.
- To add an item to the top of the stack, use append().
- To retrieve an item from the top of the stack use pop().

3.1.2 Using Lists as Queues

- It is also possible to use a list as a queue, where the first element added is the first element retrieved **("first-in, first-out")**.
- While appends and pops from the end of the list are fast, doing inserts or pops from the beginning of a list is slow.
- To implement a queue, use collections.deque which was designed to have fast appends and pops from both ends.

3.1.3 Lists Comprehensions

- List Comprehensions provide a concise way to create lists.
- Common applications are to make new lists where each element is the result of some operations, applied to each member of another sequence or iterable.
- **Example:**
```
squares = [ ]
for x in range(10):
    squares.append(x**2)
squares
```
Output: [0,1,4,9,16,25,36,49,64,81]

- We can calculate the list of squares without any side effects using:
- **Example:** squares = list(map(lambda x:x**2, range(10)))

OR

- **Example:** squares = [x**2 for x in range(10)]

Output: [0,1,4,9,16,25,36,49,64,81]

- A list comprehension consists of brackets containing an expression followed by a for clause, then zero or more for or if clauses.

- **Example:** [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
  Output: [(1,3), (1,4), (2,3), (2,1), (2,4), (3,1), (3,4)]

OR

- **Example:** combs = [ ]
  for x in [1,2,3]:
  for y in [3,1,4]:
  if x != y:
  combs.append((x, y))
  combs

Output: [(1,3), (1,4), (2,3), (2,1), (2,4), (3,1), (3,4)]

3.1.4 Nested List Comprehensions

- The initial expression in a list comprehension can be any arbitrary expression, including another list comprehension.
- Consider the following example of a 3x4 matrix implemented as a list of 3 lists of length 4:
- **Example:** matrix = [[1,2,3,4]],
  [5,6,7,8],
  [9,10,11,12]]

The following list comprehension will transpose rows and columns:

[[row[i] for row in matrix] for i in range(4)]

Output: [[1,5,9],[2,6,10],[3,7,11],[4,8,12]]

3.2 The del statement

- There is a way to remove an item from a list given its index instead of its value: the del statement.
- This differs from the pop() method which returns a value.
- The del statement can also be used to remove slices from list or clear the entire list.
- **Example:** a = [1,2,4,6,8]
  del a[0]
  a

Output: [2,4,6,8]

```
del a[1:3]
a
```

Output: [2,8]

- del can also be used to del entire variables.

3.3 Tuples and Sequences

- A tuple consists of a number of values separated by commas, for instance.
- Though tuples may seem similar to lists, they are often used in different situations and for different purposes.
- Tuples are immutable, and usually contain a heterogeneous sequence of elements.
- **Example:**

```
t = 12345, 54321, 'hello!'
t[0]
```

Output: 12345

- Empty tuples are constructed by an empty pair of parentheses; a tuple with one item is constructed by following a value with a comma.
- **Example:**

```
empty = ( )                    singleton = 'hello'
len(empty)                     len(singleton)
```

Output: 0 1

3.4 Sets

- Python also includes a data type for sets.
- A set is an unordered collection with no duplicate elements.
- Curly braces or the set( ) function can be used to create sets. **Note:** To create an empty set you have to use set( ), not { }.
- Set objects also support mathematical operations like union, intersection, difference and symmetric difference.
- **Example:**

```
basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
print(basket)
```

Output: {'orange', 'banana', 'pear', 'apple'}

3.5 Dictionaries

- Python dictionary is an unordered collection of items.

- It is best to think of dictionary as a set of key : value pairs, with the requirement that values can be of any data type and can repeat, keys must be of immutable type and must be unique.
- Creating a dictionary is as simple as placing items inside curly braces { } separated by commas.
- **Example:** dict = {'jack':4098, 'sape':4139}
  dict['guido'] = 4127
  dict
  Output: {'jack':4098, 'sape':4139, 'guido':4127}

3.6 Looping Techniques

- When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the items( ) method.
  Example: knights = {'gallahad':'the pure', 'robin':'the brave'}
  for k, v in knights.item( ):
  print(k,v)
  Output: gallahad the pure
  robin the brave

- When looping through a sequence, the position index and corresponding value can be retrieved at the same time using the enumerate( ) function.
  Example: for i, v in enumerate(['tic', 'tac', 'toe'])
  print(i,v)
  Output: 0 tic
  1 tac
  2 toe

- To loop over two or more sequences at the same time, the entries can be paired with the zip( ) function.
  Example: numberlist = [1,2,3]
  str_list = ['one', 'two', 'three']
  result=zip(n, s)`
  result_set = set(result)
  print(result_set)
  Output: {[1, 'one'), (2, 'two'), (3, 'three')}

- To loop over a sequence in reverse, first specify the sequence in a forward direction and then call the reversed( ) function.

- **Example:** for i in reversed(range(1, 10, 2)):
  print(i)

  Output: 9
  7
  5
  3
  1

- To loop over a sequence in sorted order, use the sorted( ) function which returns a new sorted list while leaving the source unaltered.

  Example: basket = [‘apple’, ‘orange’, ‘apple’, ‘pear’, ‘orange’, ‘banana’]
  for f in sorted(set(basket)):
  print(f)

  Output: apple
  banana
  orange
  pear

4. MODULES:

If you quit from the Python interpreter and enter it again, the definitions you have made are lost. Therefore, if you want to write somewhat longer program, you are better off using a text editor to prepare the input for the interpreter and running it with that file as input instead. This is known as creating a script.

To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a module.

- A module is a file containing Python definitions and statements.
- A file containing Python code, for example***:*** **example.py ,** is called a module, and its module name would be **example.**
- **Example:** #Fibonacci numbers module

```
def fib(n):      #write Fibonacci series upto n
  a , b = 0 , 1
  while a < n:
     print(a, end=' ')
     a , b = b , a+b
  print()


def fib2(n):      #return Fibonacci series upto n
 result = []
 a , b = 0 , 1
while a < n:
     result.append(a)
     a , b = b , a+b
 return result
```

4.1 More on Modules

- A module can contain executable statements as well as function definitions. These statements are intended to initialize the module. They are executed only the first time the module name is encountered in an import statement.
- Each module has its own private symbol table, which is used as the global symbol table by all functions defined in the module. Thus, the author of a module can use global variables in the module without worrying about accidental clashes with a user's global variable.
- Modules can import other modules. It is customary but not required to place all **import** statements at the beginning of the module.

- There is a variant of the **import** statement that imports names from a module directly into the importing module's symbol table.
  Example: from fibo import fib, fib2
  fib(500)

- This does not introduce the module name from which the imports are taken in the local symbol table (so in the example, fibo is not defined).
- There is even a variant to import all names that a module defines:
  Example: import fibo import *
  fibo(500)
  Output: 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377

- This imports all names except those beginning with an underscore (_). In most cases Python programmers do not use this facility since it introduces an unknown set of names into the interpreter, possibly hiding some things you have already defined.
- If the module name is followed by as, then the name following as is bound directly to the imported module.
- **Example:** import fibo as fib
  fib.fib(500)
  Output: 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377

4.1.1 Executing modules as scripts

- When you run a Python module with

  python fibo.py <arguments>

  the code in the module will be executed, just as if you imported it, but with the _name_ set to "_main_". That means that by adding this code at the end of your module:

  ```
  if _name_ == "_main_":
     import sys
     fib(int(sys.argv[1]))
  ```

  you can make the file usable as a script as well as an importable module, because the code that parses the command line only runs if the module is executed as the "main" file:

  $python fibo.py 50
  Output: 0 1 1 2 3 5 8 13 21 34

4.1.2 The Module Search Path

- When a module named spam is imported, the interpreter first searches for a built – in module with that name. If not found, it then searches for a file named spam.py in a list of directories given by the variable sys.path. sys.path is initialized from these locations:
  - The directory containing the input script (or the current directory when no file is specified).
  - PYTHONPATH (a list of directory names, with the same syntax as the shell variable PATH).
  - The installation – dependent default.

4.1.3 "Compiled" Python files

- To speed up loading modules, Python caches the compiled version of each module in the _pycache_ directory under the name module.version.pyc, where the version encodes the format of the format of the compiled files; it generally contains the Python version number.
- Python checks the modification date of the source against the compiled version to see if it's out of date and needs to be recompiled. This a completely automatic process.
- Python does not check the cache in two circumstances.
  First, it always recompiles and does not store the result for the module that's loaded directly from the command line.
  Second, it does not check the cache if there is no source module.

4.2 Standard Modules

- Python comes with a library of standard modules, described in a separate document, the Python Library Reference.
- Some modules are built into the interpreter; these provide access to operations that are not part of the core of the language but are nevertheless built in, either for efficiency or to provide access to operating system primitives such as system calls.

4.3 The dir( ) Function

- The built – in function dir( ) is used to find out which names a module defines. It returns a sorted list of strings.
- dir( ) function tries to return a valid list of attributes and methods of the object it is called upon. It behaves differently with different objects, as it aims to produce the most relevant data, rather than complete information.

4.4 Packages

- Packages are a way of structuring Python's module namespace by using "dotted module names". For example, the module name A.B designates a submodule named B in a package named A.
- When importing the package, Python searches through the directories on sys.path looking for the package subdirectory.
- The __init__.py files are required to make Python treat directories containing the file as packages. In the simplest case, __init__.py can just be an empty file, but it can also execute initialization code for the package or set the __all__ variable.

4.4.1 Importing * From a Package

- The import statement uses the following convention: if a package's __init__.py code defines a list named __all__, it is taken to be the list of module names that should be imported when *from package import* * is encountered.
- It is up to the package author to keep this list up-to-date when a new version of the package is released. Package authors may also decide not to support it, if they don't see a use for importing * from their package.

4.4.2 Packages in Multiple Directories

- Packages support one more special attribute, __path__. This is initialized to be a list containing the name of the directory holding the package's __init__.py before the code in that file is executed.
- This variable can be modified; doing so affects future searches for modules and subpackages contained in the package.

5. INPUT AND OUTPUT

There are several ways to present the output of a program; data can be printed in a human – readable form, or written to a file for future use.

5.1 Fancier Output Formatting

- So far we've encountered two ways of writing values: *expression statements* and the print() function. (A third way is using the write() method of file objects; the standard output file can be referenced as sys.stdout. See the Library Reference for more information on this.)
- **Example:** year = 2016
  event = 'Referendum'
  f'Rexsults of the {year} {event}'
  Output: 'Result of the 2016 Referendum'
- The str.format() method of strings requires more manual effort. You'll still use { and } to mark where a variable will be substituted and can provide detailed formatting directives, but you'll also need to provide the information to be formatted.

- When you don't need fancy output but just want a quick display of some variables for debugging purposes, you can convert any value to a string with the repr() or str() functions.
- The str() function is meant to return representations of values which are fairly human-readable, while repr() is meant to generate representations which can be read by the interpreter.

5.1.1 Formatted String Literals

- A *formatted string literal* or *f-string* is a string literal that is prefixed with 'f' or 'F'. These strings may contain replacement fields, which are expressions delimited by curly braces {}. While other string literals always have a constant value, formatted strings are really expressions evaluated at run time.
- An optional format specifier can follow the expression. This allows greater control over how the value is formatted. The following example rounds pi to three places after the decimal:
- **Example:** import math
  print(f'The value of pi is approximately {math.pi: .3f}.')
  Output: The value of pi approximately 3.142
  Passing an integer after the ':' will cause that field to be a minimum number of characters wide. This is useful for making columns line up.

- Other modifiers can be used to convert the value before it is formatted. '!a' applies ascii(), '!s' applies str(), and '!r' applies repr():

5.1.2 The String format( ) Method

- Basic usage of the str.format() method looks like this:

Example: print('We are the {} who say "{}!"'.format('knights' , 'Ni'))

Output: We are the knights who say "Ni!"

- The brackets and characters within them (called format fields) are replaced with the objects passed into the str.format() method. A number in the brackets can be used to refer to the position of the object passed into the str.format() method.
- If keyword arguments are used in the str.format() method, their values are referred to by using the name of the argument.
  Example: print('This {food} is {adjective}. ' .format(food='spam', adjective='absolutely horrible'))
  Output: This spam is absolutely horrible.

- If you have a really long format string that you don't want to split up, it would be nice if you could reference the variables to be formatted by name instead of by position. This can be done by simply passing the dict and using square brackets '[]' to access the keys.
  Example: table = {'Sjoerd': 4127, 'Jack': 4098, 'Dabc': 8637678}
  print('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d};' 'Dcab: {0[Dcab]:d}'.format(table))
  Output: Jack: 4098; Sjoerd: 4127; Dcab: 8637678

5.1.3 Manual String Formatting

- Here's the table of squares and cubes, formatted manually:
  Example: for x in range(1,11):
  print(repr(x).rjust(2), repr(x*x).rjust(3), end=' ')
  # Note use of 'end' on previous line
  print(repr(x*x*x).rjust(4))
  Output:
  1 1 1
  2 4 8
  3 9 27
  4 16 64
  5 25 125
  6 36 216
  7 49 343

```
 8  64  512
 9  81  729
10 100 1000
```

- The str.rjust() method of string objects right-justifies a string in a field of a given width by padding it with spaces on the left. There are similar methods str.ljust() and str.center().
- These methods do not write anything, they just return a new string. If the input string is too long, they don't truncate it, but return it unchanged; this will mess up your column lay-out but that's usually better than the alternative, which would be lying about a value.

5.1.4 Old String Formatting

- The % operator (modulo) can also be used for string formatting. Given 'string' % values, instances of % in string are replaced with zero or more elements of values. This operation is commonly known as string interpolation.
  Example: import math
  print('The value of pi is approximately %5.3f.' %math.pi)
  Output: The value of pi approximately 3.142

5.2 Reading and Writing Files

- open() returns a file object, and is most commonly used with two arguments: open(filename, mode).
  f = open('workfile' , 'w')

- The first argument is a string containing the filename. The second argument is another string containing a few characters describing the way in which the file will be used.
- *mode* can be 'r' when the file will only be read.
  'w' for only writing (an existing file with the same name will be erased). **and**
  'a' opens the file for appending; any data written to the file is automatically added to the end.
  'r+' opens the file for both reading and writing.

- Normally, files are opened in *text mode*, that means, you read and write strings from and to the file, which are encoded in a specific encoding. If encoding is not specified, the default is platform dependent (see open()). 'b' appended to the mode opens the file in *binary mode*: now the data is read and written in the form of bytes objects. This mode should be used for all files that don't contain text.
- It is good practice to use the with keyword when dealing with file objects. The advantage is that the file is properly closed after its suite finishes, even if an exception is raised at some point. Using with is also much shorter than writing equivalent try-finally blocks:

Example: with open('workfile') as f:read_data = f.read()

 # We can check that the file has been automatically closed.
f.closed
Output: True

5.2.1 Methods of File Objects

- To read a file's contents, call f.read(size), which reads some quantity of data and returns it as a string (in text mode) or bytes object (in binary mode). *size* is an optional numeric argument. When *size* is omitted or negative, the entire contents of the file will be read and returned; it's your problem if the file is twice as large as your machine's memory.
  If the end of the file has been reached, f.read() will return an empty string ('').
  Example: f.read()
  Output: 'This is the entire file.\n'
  Example: f.read()
  Output: ''

- f.readline() reads a single line from the file; a newline character (\n) is left at the end of the string, and is only omitted on the last line of the file if the file doesn't end in a newline.
  This makes the return value unambiguous; if f.readline() returns an empty string, the end of the file has been reached, while a blank line is represented by '\n', a string containing only a single newline.
  Example: f.readline()
  Output: 'This is the first line of the file.\n'

  Example: f.readline()
  Output: 'Second line of the file\n'

  Example: f.readline()
  Output: ''
- For reading lines from a file, you can loop over the file object. This is memory efficient, fast, and leads to simple code:
  Example: for line in f:
  print(line, end='')
  Output: This is the first line of the file.
  Second line of the file

- f.write(string) writes the contents of *string* to the file, returning the number of characters written.
  Example: f.write('This is a test\n')
  Output: 15
- Other types of objects need to be converted – either to a string (in text mode) or a bytes object (in binary mode) – before writing them:
  Example: value = ('the answer', 42)
  s = str(value) *# convert the tuple to string*
  f.write(s)
  Output: 18
- f.tell() returns an integer giving the file object's current position in the file represented as number of bytes from the beginning of the file when in binary mode and an opaque number when in text mode.
- To change the file object's position, use f.seek(offset, whence).
  Example: f = open('workfile', 'rb+')
  f.write(b'0123456789abcdef')
  Output:16

  Example: f.seek(5) *# Go to the 6th byte in the file*
  Output: 5

  Example: f.read(1)
  Output: b'5'

  Example: f.seek(-3, 2) *# Go to the 3rd byte before the end*
  Output:13

  Example:f.read(1)
  Output: b'd'

5.2.2 Saving structured data with json

- Strings can easily be written to and read from a file. Numbers take a bit more effort, since the read() method only returns strings, which will have to be passed to a function like int(), which takes a string like '123' and returns its numeric value 123.
  When you want to save more complex data types like nested lists and dictionaries, parsing and serializing by hand becomes complicated.

- Rather than having users constantly writing and debugging code to save complicated data types to files, Python allows you to use the popular data interchange format called JSON (JavaScript Object Notation).
  The standard module called json can take Python data hierarchies, and convert them to string representations; this process is called *serializing*. Reconstructing the data from the string representation is called *deserializing*. Between serializing and deserializing, the string representing the object may have been stored in a file or data, or sent over a network connection to some distant machine.

- If you have an object x, you can view its JSON string representation with a simple line of code:
  Example: import json
  json.dumps([1, 'simple', 'list'])
  Output: '[1, "simple", "list"]'
- Another variant of the dumps() function, called dump(), simply serializes the object to a text file. So if f is a text file object opened for writing, we can do this:
  json.dump(x, f)

- To decode the object again, if f is a text file object which has been opened for reading:
  x = json.load(f)

6. ERRORS AND EXCEPTIONS

- There are (at least) two distinguishable kinds of errors: *syntax errors* and *exceptions*.

6.1 Syntax Errors

- Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python:

```
Example: while True print('Hello world')
       File "<stdin>", line 1
       while True print('Hello world')
                      ^
SyntaxError: invalid syntax
```

- The parser repeats the offending line and displays a little 'arrow' pointing at the earliest point in the line where the error was detected. The error is caused by (or at least detected at) the token *preceding* the arrow: in the example, the error is detected at the function print(), since a colon (':') is missing before it. File name and line number are printed so you know where to look in case the input came from a script.

6.2 Exceptions

- Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called *exceptions* and are not unconditionally fatal:

```
Example: 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

- The last line of the error message indicates what happened. Exceptions come in different types, and the type is printed as part of the message: the types in the example are ZeroDivisionError, NameError and TypeError.

6.3 Handling Exceptions

- An exception is an event which occurs during the execution of a program.
- It is possible to write programs that handle selected exceptions.

```
Example: while True:
...      try:
...          x = int(input("Please enter a number: "))
...          break
...      except ValueError:
...          print("Oops!  That was no valid number.  Try again...")
...
```

The **try** statement works as follows:

- 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.

6.4 Raising Exceptions

- The raise statement allows the programmer to force a specified exception to occur.

```
Example: raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: HiThere
```

- The sole argument to raise indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from Exception). If an exception class is passed, it will be implicitly instantiated by calling its constructor with no arguments:

- If you need to determine whether an exception was raised but don't intend to handle it, a simpler form of the raise statement allows you to re-raise the exception:

```
Example: try:
...         raise NameError('HiThere')
...     except NameError:
...         print('An exception flew by!')
...         raise
...
An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: HiThere
```

6.5 User – defined Exceptions

- Programs may name their own exceptions by creating a new exception class. Exceptions should typically be derived from the Exception class, either directly or indirectly.
- Exception classes can be defined which do anything any other class can do, but are usually kept simple, often only offering a number of attributes that allow information about the error to be extracted by handlers for the exception.
- When creating a module that can raise several distinct errors, a common practice is to create a base class for exceptions defined by that module, and subclass that to create specific exception classes for different error conditions:

```
Example:
class Error(Exception):
    """Base class for exceptions in this module."""
    pass

class InputError(Error):
    """Exception raised for errors in the input.

    Attributes:
        expression -- input expression in which the error occurred
        message -- explanation of the error
    """

    def __init__(self, expression, message):
        self.expression = expression
```

```
        self.message = message

class TransitionError(Error):
    """Raised when an operation attempts a state transition that's
not
    allowed.

    Attributes:
        previous -- state at beginning of transition
        next -- attempted new state
        message -- explanation of why the specific transition is
not allowed
    """

    def __init__(self, previous, next, message):
        self.previous = previous
        self.next = next
        self.message = message
```

- Most exceptions are defined with names that end in "Error", similar to the naming of the standard exceptions.

6.6 Defining Clean – up Actions

- The try statement has another optional clause which is intended to define clean-up actions that must be executed under all circumstances.

```
Example: try:
...      raise KeyboardInterrupt
...    finally:
...      print('Goodbye, world!')
...
Goodbye, world!
KeyboardInterrupt
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
```

- If a finally clause is present, the finally clause will execute as the last task before the try statement completes. The finally clause runs whether or not the try statement produces an exception. The following points discuss more complex cases when an exception occurs:

- If an exception occurs during execution of the try clause, the exception may be handled by an except clause. If the exception is not handled by an except clause, the exception is re-raised after the finally clause has been executed.
- An exception could occur during execution of an except or else clause. Again, the exception is re-raised after the finally clause has been executed.
- If the try statement reaches a break, continue or return statement, the finally clause will execute just prior to the break, continue or return statement's execution.
- If a finally clause includes a return statement, the returned value will be the one from the finally clause's return statement, not the value from the try clause's return statement.

```
Example: def bool_return():
...      try:
...          return True
...      finally:
...          return False
...
>>> bool_return()
False
```

6.7 Predefined Clean – up Actions

- Some objects define standard clean-up actions to be undertaken when the object is no longer needed, regardless of whether or not the operation using the object succeeded or failed.

```
Example: for line in open("myfile.txt"):
             print(line, end="")
```

The problem with this code is that it leaves the file open for an indeterminate amount of time after this part of the code has finished executing. This is not an issue in simple scripts, but can be a problem for larger applications.

- The with statement allows objects like files to be used in a way that ensures they are always cleaned up promptly and correctly.

```
Example: with open("myfile.txt") as f:
```

```
        for line in f:
            print(line, end="")
```

After the statement is executed, the file *f* is always closed, even if a problem was encountered while processing the lines. Objects which, like files, provide predefined clean-up actions will indicate this in their documentation.

7. CLASSES

- Classes provide a means of bundling data and functionality together. Creating a new class creates a new *type* of object, allowing new *instances* of that type to be made.
  Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state.
- Compared with other programming languages, Python's class mechanism adds classes with a minimum of new syntax and semantics. It is a mixture of the class mechanisms found in C++ and Modula-3.
- Python classes provide all the standard features of Object Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name.

7.1 A Word About Names and Objects

- Objects have individuality, and multiple names (in multiple scopes) can be bound to the same object. This is known as aliasing in other languages. This is usually not appreciated on a first glance at Python, and can be safely ignored when dealing with immutable basic types (numbers, strings, tuples).
- However, aliasing has a possibly surprising effect on the semantics of Python code involving mutable objects such as lists, dictionaries, and most other types. This is usually used to the benefit of the program, since aliases behave like pointers in some respects.
- For example, passing an object is cheap since only a pointer is passed by the implementation; and if a function modifies an object passed as an argument, the caller will see the change — this eliminates the need for two different argument passing mechanisms as in Pascal.

7.2 Python Scopes and Namespaces

- A *namespace* is a mapping from names to objects. Most namespaces are currently implemented as Python dictionaries, but that's normally not noticeable in any way (except for performance), and it may change in the future. Examples of namespaces are: the set of built-in names (containing functions such as abs(), and built-in exception names); the global names in a module; and the local names in a function invocation.
- Namespaces are created at different moments and have different lifetimes. The namespace containing the built-in names is created when the Python interpreter starts up, and is never deleted.
  - The global namespace for a module is created when the module definition is read in; normally, module namespaces also last until the interpreter quits.

- The local namespace for a function is created when the function is called, and deleted when the function returns or raises an exception that is not handled within the function.

➢ A *scope* is a textual region of a Python program where a namespace is directly accessible. "Directly accessible" here means that an unqualified reference to a name attempts to find the name in the namespace.

➢ Although scopes are determined statically, they are used dynamically. At any time during execution, there are 3 or 4 nested scopes whose namespaces are directly accessible:

- the innermost scope, which is searched first, contains the local names.
- the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contains non-local, but also non-global names.
- the next-to-last scope contains the current module's global names.
- the outermost scope (searched last) is the namespace containing built-in names.

➢ If a name is declared global, then all references and assignments go directly to the middle scope containing the module's global names.

7.2.1 Scopes and Namespaces Example

➢ This is an example demonstrating how to reference the different scopes and namespaces, and how global and nonlocal affect variable binding:

Example:

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
```

```
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

The output of the example code is:

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

7.3 A First Look at Classes

Classes introduce a little bit of new syntax, three new object types, and some new semantics.

7.3.1 Class Definition Syntax

- The simplest form of class definition looks like this:

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

- Class definitions, like function definitions (def statements) must be executed before they have any effect.
- In practice, the statements inside a class definition will usually be function definitions, but other statements are allowed, and sometimes useful — we'll come back to this later.
- When a class definition is entered, a new namespace is created, and used as the local scope — thus, all assignments to local variables go into this new namespace.
- When a class definition is left normally (via the end), a *class object* is created. This is basically a wrapper around the contents of the namespace created by the class definition; we'll learn more about class objects in the next section.

7.3.2 Class Objects

- Class objects support two kinds of operations: attribute references and instantiation.
- *Attribute references* use the standard syntax used for all attribute references in Python: obj.name. Valid attribute names are all the names that were in the class's namespace when the class object was created. So, if the class definition looked like this:

Example:

```
class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
        return 'hello world'
```

then MyClass.i and MyClass.f are valid attribute references, returning an integer and a function object, respectively. Class attributes can also be assigned to, so you can change the value of MyClass.i by assignment. __doc__ is also a valid attribute, returning the docstring belonging to the class: "A simple example class".

- Class *instantiation* uses function notation. Just pretend that the class object is a parameterless function that returns a new instance of the class. For example (assuming the above class):

```
x = MyClass()
```

- creates a new *instance* of the class and assigns this object to the local variable x.

- The instantiation operation ("calling" a class object) creates an empty object. Many classes like to create objects with instances customized to a specific initial state. Therefore a class may define a special method named __init__(), like this:

Example:

```
def __init__(self):
    self.data = []
```

- When a class defines an __init__() method, class instantiation automatically invokes __init__() for the newly-created class instance. So in this example, a new, initialized instance can be obtained by:

```
x = MyClass()
```

7.3.3 Instance Objects

- The only operations understood by instance objects are attribute references. There are two kinds of valid attribute names: data attributes and methods.
- *data attributes* correspond to "instance variables" in Smalltalk, and to "data members" in C++. Data attributes need not be declared; like local variables, they spring into existence when they are first assigned to. For example, if x is the instance of MyClass created above, the following piece of code will print the value 16, without leaving a trace:

Example:

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print(x.counter)
del x.counter
```

- The other kind of instance attribute reference is a *method*. A method is a function that "belongs to" an object. (In Python, the term method is not unique to class instances: other object types can have methods as well.
- Valid method names of an instance object depend on its class. By definition, all attributes of a class that are function objects define corresponding methods of its instances. So in our example, x.f is a valid method reference, since MyClass.f is a function, but x.i is not, since MyClass.i is not. But x.f is not the same thing as MyClass.f — it is a *method object*, not a function object.

7.3.4 Method Objects

- Usually, a method is called right after it is bound:

```
x.f()
```

- In the MyClass example, this will return the string 'hello world'. However, it is not necessary to call a method right away: x.f is a method object, and can be stored away and called at a later time.
  Example:

```
xf = x.f
```

```
while True:
    print(xf())
```

will continue to print hello world until the end of time.

- What exactly happens when a method is called? You may have noticed that x.f() was called without an argument above, even though the function definition for f() specified an argument. What happened to the argument? Surely Python raises an exception when a function that requires an argument is called without any — even if the argument isn't actually used…
- Actually, you may have guessed the answer: the special thing about methods is that the instance object is passed as the first argument of the function. In our example, the call x.f() is exactly equivalent to MyClass.f(x).

7.3.5 Class and Instance Variables

- Generally speaking, instance variables are for data unique to each instance and class variables are for attributes and methods shared by all instances of the class:
  Example:

```
class Dog:

    kind = 'canine'         # class variable shared by all
instances

    def __init__(self, name):
        self.name = name    # instance variable unique to each
instance

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind                  # shared by all dogs
'canine'
>>> e.kind                  # shared by all dogs
'canine'
>>> d.name                  # unique to d
'Fido'
>>> e.name                  # unique to e
'Buddy'
```

- As discussed in A Word About Names and Objects, shared data can have possibly surprising effects with involving mutable objects such as lists and dictionaries. For example, the *tricks* list in the following code should not be used as a class variable because just a single list would be shared by all *Dog* instances:

```
class Dog:

    tricks = []             # mistaken use of a class variable

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks                # unexpectedly shared by all dogs
['roll over', 'play dead']
```

- Correct design of the class should use an instance variable instead:

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list for each
dog

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
```

```
['roll over']
>>> e.tricks
['play dead']
```

7.4 Random Remarks

- If the same attribute name occurs in both an instance and in a class, then attribute lookup prioritizes the instance:
  Example:

```
>>> class Warehouse:
        purpose = 'storage'
        region = 'west'

>>> w1 = Warehouse()
>>> print(w1.purpose, w1.region)
storage west
>>> w2 = Warehouse()
>>> w2.region = 'east'
>>> print(w2.purpose, w2.region)
storage east
```

- Data attributes may be referenced by methods as well as by ordinary users ("clients") of an object. In other words, classes are not usable to implement pure abstract data types. In fact, nothing in Python makes it possible to enforce data hiding — it is all based upon convention.

- There is no shorthand for referencing data attributes (or other methods!) from within methods. I find that this actually increases the readability of methods: there is no chance of confusing local variables and instance variables when glancing through a method.

- Any function object that is a class attribute defines a method for instances of that class. It is not necessary that the function definition is textually enclosed in the class definition: assigning a function object to a local variable in the class is also ok.
  Example:

```
# Function defined outside the class
def f1(self, x, y):
    return min(x, x+y)
```

```
class C:
    f = f1

    def g(self):
        return 'hello world'

    h = g
```

- Now f, g and h are all attributes of class C that refer to function objects, and consequently they are all methods of instances of C — h being exactly equivalent to g.

- Methods may call other methods by using method attributes of the self argument:
  Example:

```
class Bag:
    def __init__(self):
        self.data = []

    def add(self, x):
        self.data.append(x)

    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

- Methods may reference global names in the same way as ordinary functions. The global scope associated with a method is the module containing its definition.

7.5 Inheritance

- Of course, a language feature would not be worthy of the name "class" without supporting inheritance. The syntax for a derived class definition looks like this:

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

- The name BaseClassName must be defined in a scope containing the derived class definition. In place of a base class name, other arbitrary expressions are also allowed. This can be useful, for example, when the base class is defined in another module:

```
class DerivedClassName(modname.BaseClassName):
```

- Execution of a derived class definition proceeds the same as for a base class. When the class object is constructed, the base class is remembered. This is used for resolving attribute references: if a requested attribute is not found in the class, the search proceeds to look in the base class.
- There's nothing special about instantiation of derived classes: DerivedClassName() creates a new instance of the class.
- Derived classes may override methods of their base classes. Because methods have no special privileges when calling other methods of the same object, a method of a base class that calls another method defined in the same base class may end up calling a method of a derived class that overrides it.

- Python has two built-in functions that work with inheritance:

  - Use isinstance() to check an instance's type: isinstance(obj, int) will be True only if obj.__class__ is int or some class derived from int.
  - Use issubclass() to check class inheritance: issubclass(bool, int) is True since bool is a subclass of int. However, issubclass(float, int) is False since float is not a subclass of int.

7.5.1 Multiple Inheritance

- Python supports a form of multiple inheritance as well. A class definition with multiple base classes looks like this:

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

- For most purposes, in the simplest cases, you can think of the search for attributes inherited from a parent class as depth-first, left-to-right, not searching twice in the same class where there is an overlap in the hierarchy. Thus, if an attribute is not found in DerivedClassName, it is searched for in Base1, then (recursively) in the base classes of Base1, and if it was not found there, it was searched for in Base2, and so on.
- In fact, it is slightly more complex than that; the method resolution order changes dynamically to support cooperative calls to super(). This approach is known in some other multiple-inheritance languages as call-next-method and is more powerful than the super call found in single-inheritance languages.

7.6 Private Variables

- "Private" instance variables that cannot be accessed except from inside an object don't exist in Python. However, there is a convention that is followed by most Python code: a name prefixed with an underscore (e.g. `_spam`) should be treated as a non-public part of the API (whether it is a function, a method or a data member).
- Since there is a valid use-case for class-private members, there is limited support for such a mechanism, called *name mangling*.
  Any identifier of the form `__spam` (at least two leading underscores, at most one trailing underscore) is textually replaced with `_classname__spam`, where `classname` is the current class name with leading underscore(s) stripped. This mangling is done without regard to the syntactic position of the identifier, as long as it occurs within the definition of a class.

- Name mangling is helpful for letting subclasses override methods without breaking intraclass method calls.
  Example:

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

    __update = update   # private copy of original update() method

class MappingSubclass(Mapping):
```

```
    def update(self, keys, values):
        # provides new signature for update()
        # but does not break __init__()
        for item in zip(keys, values):
            self.items_list.append(item)
```

- The above example would work even if MappingSubclass were to introduce a __update identifier since it is replaced with _Mapping__update in the Mapping class and _MappingSubclass__update in the MappingSubclass class respectively.

7.7 Odds and Ends

- Sometimes it is useful to have a data type similar to the Pascal "record" or C "struct", bundling together a few named data items. An empty class definition will do nicely:
  Example:

```
class Employee:
    pass

john = Employee()  # Create an empty employee record

# Fill the fields of the record
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

- A piece of Python code that expects a particular abstract data type can often be passed a class that emulates the methods of that data type instead. For instance, if you have a function that formats some data from a file object, you can define a class with methods read() and readline() that get the data from a string buffer instead, and pass it as an argument.
- Instance method objects have attributes, too: m.__self__ is the instance object with the method m(), and m.__func__ is the function object corresponding to the method.

7.8 Iterators

- Iterator is an object.
- It remembers its state i.e. where it is during iteration.
- **__iter__()** method initializes an iterator.

- It has a **__next__()** method which returns the next item in iteration and points to the next element.
- It is also self iterable.
- Iterators are objects with which we can iterate over iterable objects like lists, strings, etc.

- The use of iterators pervades and unifies Python. Behind the scenes, the for statement calls iter() on the container object. The function returns an iterator object that defines the method __next__() which accesses elements in the container one at a time.
  When there are no more elements, __next__() raises a StopIteration exception which tells the for loop to terminate. You can call the __next__() method using the next() built-in function; this example shows how it all works:
  Example:

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> next(it)
'a'
>>> next(it)
'b'
>>> next(it)
'c'
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    next(it)
StopIteration
```

7.9 Generators

- Generators allows the functions to behave like iterators.
- Generators are simple and powerful tools for creating iterators.
- Generators are function that return an iterable collection of items, one at a time, in a set manner.
- Generators, in general are used to create iterators with a different approach. They employ the use of yield keyword rather then return to return the generator objects
- An example shows that generators can be trivially easy to create:

```
def reverse(data):
```

```
    for index in range(len(data)-1, -1, -1):
        yield data[index]
```

```
>>>
```

```
>>> for char in reverse('golf'):
...     print(char)
...
f
l
o
g
```

7.10 Generator Expressions

- Some simple generators can be coded succinctly as expressions using a syntax similar to list comprehensions but with parentheses instead of square brackets. These expressions are designed for situations where the generator is used right away by an enclosing function.
- Generator expressions are more compact but less versatile than full generator definitions and tend to be more memory friendly than equivalent list comprehensions.
  Example:

```
>>> sum(i*i for i in range(10))                 # sum of squares
285

>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvec, yvec))         # dot product
260

>>> unique_words = set(word for line in page  for word in
line.split())

>>> valedictorian = max((student.gpa, student.name) for student in
graduates)

>>> data = 'golf'
>>> list(data[i] for i in range(len(data)-1, -1, -1))
['f', 'l', 'o', 'g']
```