Python Interpreter:
Python is formally an interpreted language. Commands are executed through a piece of software known as the Python interpreter. The interpreter receives a command, evaluates that command, and reports the result of the command.

Objects in Python:
Python is an object-oriented language and classes form the basis for all data types.

Instantiation:
The process of creating a new instance of a class is known as instantiation. In general, the syntax for instantiating an object is to invoke the constructor of a class. For example, if there were a class named Widget, we could create an instance of that class using a syntax such as w = Widget( ), assuming that the constructor does not require any parameters. If the constructor does require parameters, we might use a syntax such as Widget(a, b, c) to construct a new instance.

Calling methods:
Python supports traditional functions that are invoked with a syntax such as sorted(data), in which case data is a parameter sent to the function. Python's classes may also define one or more methods (also known as member functions), which are invoked on a specific instance of a class using the dot (".") operator. For example, Python's list class has a method named sort that can be invoked with a syntax such as data.sort( ).

Python Built-in Classes:

Class	Description	Immutable?
bool	Boolean value	Y
int	integer (arbitrary magnitude)	Y
float	floating-point number	Y
list	mutable sequence of objects	
tuple	immutable sequence of objects	Y
str	character string	Y
set	unordered set of distinct objects	
frozenset	immutable form of set class	Y
dict	associative mapping (aka dictionary)	

Operator Precedence:

	Type	**Symbols**
1	member access	expr.member
2	function/method calls	expr(...)

Logical Operators:
Python supports the following keyword operators for Boolean values:
not unary negation
and conditional and
or conditional or

Equality Operators:
Python supports the following operators to test two notions of equality:
is - same identity
is not - different identity
== - equivalent
!= - not equivalent

Comparison Operators:
Data types may define a natural order via the following operators:
< less than
<= less than or equal to
> greater than
>= greater than or equal to

Arithmetic Operators:
Python supports the following arithmetic operators:
+ addition
− subtraction
* multiplication
/ true division
// integer division
% the modulo operator

Bitwise Operators:
Python provides the following bitwise operators for integers:
∼ bitwise complement (prefix unary operator)
& bitwise and
| bitwise or
ˆ bitwise exclusive-or

	container subscripts/slices	expr[...]
3	exponentiation	**
4	unary operators	+expr, −expr, ˜expr
5	multiplication, division	*, /, //, %
6	addition, subtraction	+, −
7	bitwise shifting	<<, >>
8	bitwise-and	&
9	bitwise-xor	ˆ
10	bitwise-or	|
11	comparisons	is, is not, ==, !=, <, <=, >, >=
	containment	in, not in
12	logical-not	not expr
13	logical-and	and
14	logical-or	or
15	conditional	val1 if cond else val2
16	assignments	=, +=, −=, =, etc.

Conditionals:

```
if first condition:
  first body
elif second condition:
  second body
elif third condition:
  third body
else:
  fourth body
```

Control flow:

```
if door is closed:
  open door( )
advance( )
```

While loops:

```
while condition:
  body
```

For Loops:

```
for element in iterable:
  body # body may refer to element as an identifier
```

<< shift bits left, filling in with zeros
>> shift bits right, filling in with sign bit

Sequence Operators:

Each of Python's built-in sequence types (str, tuple, and list) support the following operator syntaxes:
s[j] element at index j
s[start:stop] slice including indices [start,stop)
s[start:stop:step] slice including indices start, start + step, start + 2 step, . . . , up to but not equalling or stop
s+t concatenation of sequences
k * s shorthand for s + s + s + ... (k times) val in s containment check
val not in s non-containment check

s == t equivalent (element by element)
s != t not equivalent
s < t lexicographically less than
s <= t lexicographically less than or equal to
s > t lexicographically greater than
s >= t lexicographically greater than or equal to

Operators for Sets and Dictionaries:

Sets and frozensets support the following operators:
key in s containment check
key not in s non-containment check
s1 == s2 s1 is equivalent to s2
s1 != s2 s1 is not equivalent to s2
s1 <= s2 s1 is subset of s2
s1 < s2 s1 is proper subset of s2
s1 >= s2 s1 is superset of s2
s1 > s2 s1 is proper superset of s2
s1 | s2 the union of s1 and s2
s1 & s2 the intersection of s1 and s2
s1 − s2 the set of elements in s1 but not s2
s1 ˆ s2 the set of elements in precisely one of s1 or s2

Dictionary:

d[key] value associated with given key
d[key] = value set (or reset) the value associated with given key
del d[key] remove key and its associated value from dictionary
key in d containment check key not in d non-containment check
d1 == d2 d1 is equivalent to d2
d1 != d2 d1 is not equivalent to d2

Extended assignments:

```
alpha = [1, 2, 3]
beta = alpha # an alias for alpha
beta += [4, 5] # extends the original list with two more elements
beta = beta + [6, 7] # reassigns beta to a new list [1, 2, 3, 4, 5, 6, 7]
print(alpha) # will be [1, 2, 3, 4, 5]
```

Functions:

```
def count(data, target):
  n=0
  for item in data:
    if item == target: # found a match
      n += 1
  return n
```

Mutable Parameters:

```
def scale(data, factor):
  for j in range(len(data)):
    data[j] = factor
```

Default Parameters:

```
def foo(a, b=15, c=27):
```

Positional parameters:
The traditional mechanism for matching the actual parameters sent by a caller, to the formal parameters declared by the function signature is based on the concept of positional arguments.

Keyword parameters:
Python supports an alternate mechanism for sending a parameter to a function known as a keyword argument. A keyword argument is specified by explicitly assigning an actual parameter to a formal parameter by name.

Python's Built-In functions:
Input/Output: print, input, and open
Character Encoding: ord and chr relate characters and their integer code points.
Mathematics: abs, divmod, pow, round, and sum provide common mathematical functionality.
Ordering: max and min apply to any data type that supports a notion of comparison, or to any collection of such values. sorted can be used to produce an ordered list of elements drawn from any existing collection.
Collections/Iterations: range generates a new sequence of numbers; len reports the length of any existing collection; functions reversed, all, any, and map operate on arbitrary iterations as well; iter and next provide a general framework for iteration through elements of a collection

Input:

```
year = int(input( In what year were you born? ))
```

Print:

```
print( Your target fat-burning heart rate is , target)
```

Files:

```
fp = open( sample.txt )
fp.read( )
```

Return the (remaining) contents of a readable file as a string

```
fp.read(k)
```

Return the next k bytes of a readable file as a string.

```
fp.readline( )
```

Return (remainder of) the current line of a readable file as a string.

```
fp.readlines( )
```

Return all (remaining) lines of a readable file as a list of strings.

```
for line in fp:
```

Iterate all (remaining) lines of a readable file.

```
fp.seek(k)
```

Common Built-In Functions:
Calling Syntax

```
abs(x)
all(iterable)
any(iterable)
chr(integer)
divmod(x, y)
hash(obj)
id(obj)
input(prompt)
isinstance(obj, cls)
iter(iterable)
len(iterable)
map(f, iter1, iter2, ...)
max(iterable)
max(a, b, c, ...)
min(iterable)
min(a, b, c, ...)
next(iterator)
open(filename, mode)
ord(char)
pow(x, y)
pow(x, y, z)
print(obj1, obj2, ...)
range(stop)
range(start, stop)
range(start, stop, step)
reversed(sequence)
round(x)
round(x, k)
sorted(iterable
sum(iterable)
type(obj)
```

Exceptional Handling:

Common Exception Types:

Exception: A base class for most error types

AttributeError: Raised by syntax obj.foo, if obj has no member

Change the current position to be at the kth byte of the file.

fp.tell( )
Return the current position, measured as byte-offset from the start.

fp.write(string)
Write given string at current position of the writable file.

fp.writelines(seq)
Write each of the strings of the given sequence at the current position of the writable file. This command does not insert any newlines, beyond those that are embedded in the strings.

print(..., file=fp)
Redirect output of print function to the file.

Raising an Exception:

```
raise ValueError( x cannot be negative )
```

Catching an Exception:

```
try:
  ratio = x / y
except ZeroDivisionError:
  ... do something else ...
```

named foo

EOFError: Raised if "end of file" reached for console or file input

IOError: Raised upon failure of I/O operation (e.g., opening file)

IndexError: Raised if index to sequence is out of bounds

KeyError: Raised if nonexistent key requested for set or dictionary

KeyboardInterrupt: Raised if user types ctrl-C while program is executing

NameError : Raised if nonexistent identifier used

StopIteration: Raised by next(iterator) if no element

TypeError: Raised when wrong type of parameter is sent to a function

ValueError : Raised when parameter has invalid value (e.g., sqrt(−5))

ZeroDivisionError: Raised when any division operator used with 0 as divisor

Iterators:

```
for element in iterable:
```

An **iterator** is an object that manages an iteration through a series of values. If variable, i, identifies an iterator object, then each call to the built-in function, next(i), produces a subsequent element from the underlying series, with a StopIteration exception raised to indicate that there are no further elements.

An **iterable** is an object, obj, that produces an iterator via the syntax iter(obj).

Conditional Expression:

```
expr1 if condition else expr2
```

Comprehension Syntax:

```
[ k k for k in range(1, n+1) ] list comprehension
{ k k for k in range(1, n+1) } set comprehension
( k k for k in range(1, n+1) ) generator comprehension
{ k:k k for k in range(1, n+1) } dictionary comprehension
```

Generators:
Most convenient technique for creating iterators in Python is through the use of generators. A generator is implemented with a syntax that is very similar to a function, but instead of returning values, a yield statement is executed to indicate each element of the series.

```
def factors(n): # traditional function that computes factors
  results = [ ] # store factors in a new list
  for k in range(1,n+1):
    if n % k == 0: # divides evenly, thus k is a factor
      results.append(k) # add k to the list of factors
  return results # return the entire list

def factors(n): # generator that computes factors
  for k in range(1,n+1):
    if n % k == 0: # divides evenly, thus k is a factor
      yield k # yield this factor as next result
```

Packaging and unpackaging:

```
data = 2, 4, 6, 8
```

results in identifier, data, being assigned to the tuple (2, 4, 6, 8). This behavior is called automatic packing of a tuple.

First class objects:
In the terminology of programming languages, first-class objects are instances of a type that can be assigned to an identifier, passed as a parameter, or returned by a function.

Modules and import statements:

```
from math import pi, sqrt
```

Creating a New Module:
To create a new module, one simply has to put the relevant

```
import math
```

Existing Modules:

- array: Provides compact array storage for primitive types.
- collections: Defines additional data structures and abstract base classes involving collections of objects.
- copy: Defines general functions for making copies of objects.
- heapq: Provides heap-based priority queue functions
- math: Defines common mathematical constants and functions
- os: Provides support for interactions with the operating system
- random: Provides random number generation.
- re: Provides support for processing regular expressions.
- sys: Provides an additional level of interaction with the Python interpreter
- time: Provides support for measuring time, or delaying a program

Important Modules and Functions:

```
from collections import Counter
cnt = Counter()
list = [1,2,3,4,1,2,6,7,3,8,1]
cnt = Counter(list)
cnt = Counter({1:3,2:4})
print(cnt[1])
print(list(cnt.elements()))
print(cnt.most_common())
deduct = {1:1, 2:2}
cnt.subtract(deduct)
print(cnt)
```

```
from collections import defaultdict
nums = defaultdict(int)
nums['one'] = 1
nums['two'] = 2
print(nums['three'])  # 0
nums = defaultdict(lambda:1)
print(nums['three'])  # 1
```

```
from collections import OrderedDict
od = OrderedDict()
od['a'] = 1
od['b'] = 2
od['c'] = 3
print(od) # OrderedDict([('a', 1), ('b', 2), ('c', 3)])
for key, value in od.items():
    print(key, value)
```

```
from collections import deque
```

definitions in a file named with a .py suffix. Those definitions can be imported from any other .py file within the same project directory. For example, if we were to put the definition of our count function into a file named utility.py, we could import that function using the syntax, from utility import count.

Pseudo-random number generation:
Python’s random module provides the ability to generate pseudo-random numbers, that is, numbers that are statistically random (but not necessarily truly random). A pseudo-random number generator uses a deterministic formula to generate the www.it-ebooks.info 50 Chapter 1. Python Primer next number in a sequence based upon one or more past numbers that it has generated. Indeed, a simple yet popular pseudo-random number generator chooses its next number based solely on the most recently chosen number and some additional parameters using the following formula. next = (a*current + b) % n;

Syntax:

- seed(hashable)
  - Initializes the pseudo-random number generator based upon the hash value of the parameter
- random()
  - Returns a pseudo-random floating-point value in the interval [0.0,1.0).
- randint(a,b)
  - Returns a pseudo-random integer in the closed interval [a,b].
- randrange(start, stop, step)
  - Returns a pseudo-random integer in the standard Python range indicated by the parameters.
- choice(seq)
  - Returns an element of the given sequence chosen pseudo-randomly.
- shuffle(seq)
  - Reorders the elements of the given sequence pseudo-randomly.

```
from heapq import heappush, heappop
heap = []
data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
for item in data:
  heappush(heap, item)

ordered = []
while heap:
  ordered.append(heappop(heap))

assert sorted(data) == ordered

heapq.heapify(listForTree) # for a min heap
heapq._heapify_max(listForTree) # for a maxheap!!
```

list = ["a","b","c"]
deq = deque(list) #deque(['a', 'b', 'c'])
deq.append("d")
deq.appendleft("e")
deq.pop()
deq.popleft()
deq.clear()
deq.count("a")
 	
from collections import ChainMap
dict1 = { 'a' : 1, 'b' : 2 }
dict2 = { 'c' : 3, 'b' : 4 }
chain_map = ChainMap(dict1, dict2) #[{'b': 2, 'a': 1}, {'c': 3, 'b': 4}]
 	
from collections import namedtuple
Student = namedtuple('Student', 'fname, lname, age')
s1 = Student('John', 'Clarke', '13')
print(s1.fname)
s2 = Student._make(['Adam','joe','18'])
print(s2)
s2 = s1._asdict()
print(s2) #OrderedDict([('fname', 'John'), ('lname', 'Clarke'), ('age', '13')])
s2 = s1._replace(age='14')
	heapq.heappop(minheap) # pop from minheap
	heapq._heappop_max(maxheap) # pop from maxheap
	
	heappush
	heappop
	heappushpop
	heapify
	heapreplace
	merge
	nlargest
	nsmallest

Note: Information gathered in this document has been collected from various sources on Internet.