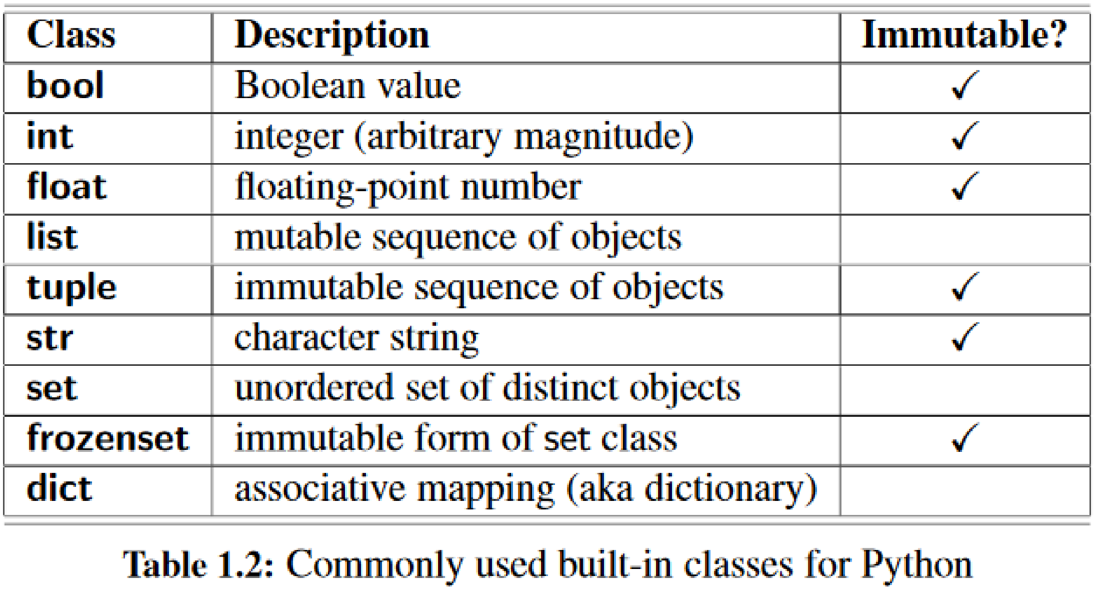
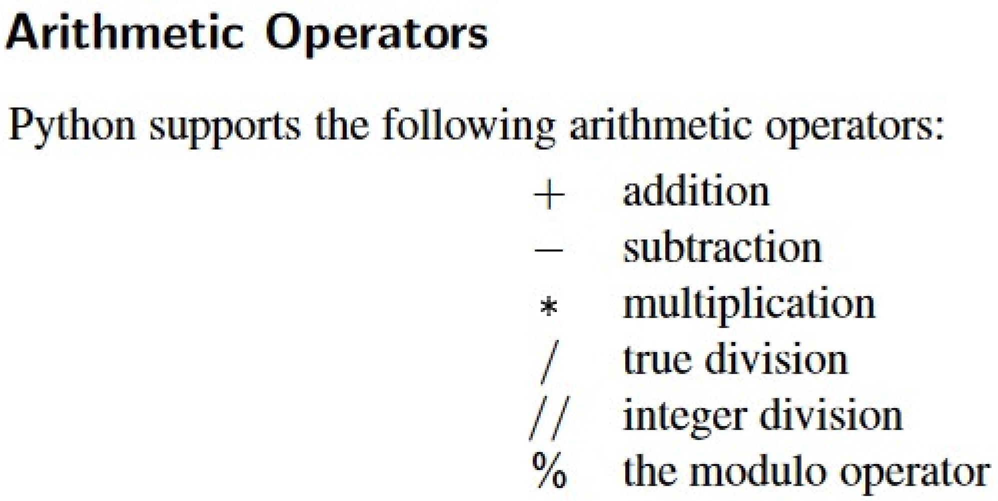
Python’s Built-In Classes



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

Table 1.2: Commonly used built-in classes for Python



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
^	bitwise exclusive-or
<<	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

Operators for Sets and Dictionaries

Sets and frozensets support the following operators:

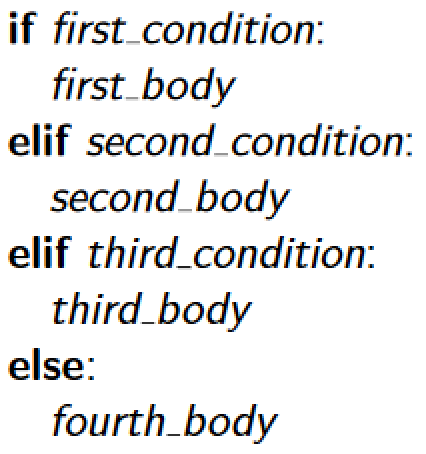
key **in** s	containment check
key **not in** s	non-containment check
s1 == s2	s1 is equivalent to s2
s1 != s2	s1 is not equivalent to s2
s1 <= s2	s1 is subset of s2
s1 < s2	s1 is proper subset of s2
s1 >= s2	s1 is superset of s2
s1 > s2	s1 is proper superset of s2
s1 | s2	the union of s1 and s2
s1 & s2	the intersection of s1 and s2
s1 − s2	the set of elements in s1 but not s2
s1 ^ s2	the set of elements in precisely one of s1 or s2

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

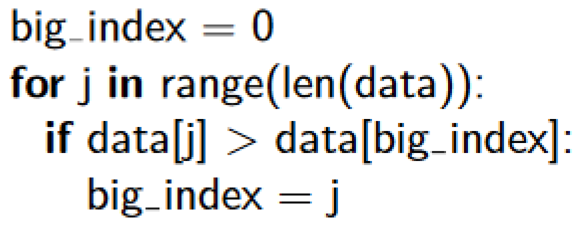
Operator Precedence

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

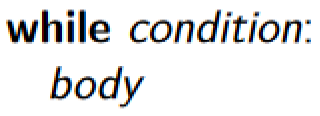


```
if first_condition:
   first_body
elif second_condition:
   second_body
elif third_condition:
   third_body
else:
   fourth_body
```



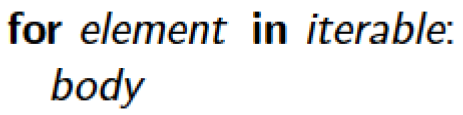
```
big_index = 0
for j in range(len(data)):
  if data[j] > data[big_index]:
     big_index = j
```

While Loops

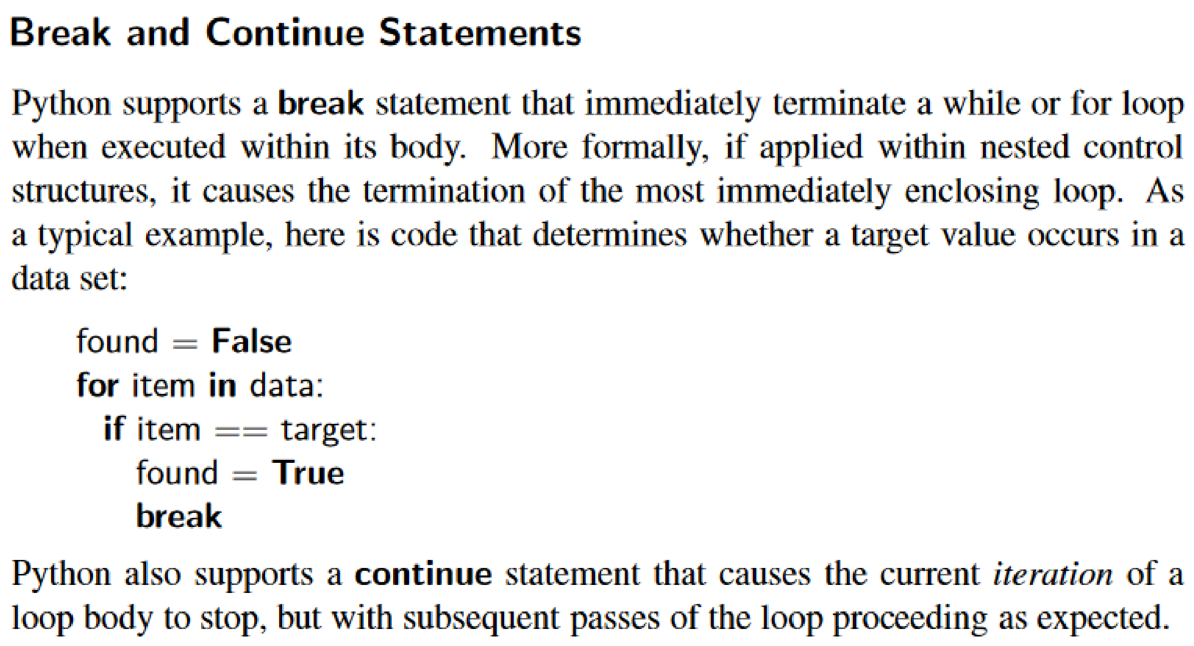


```
while condition:
   body
```

For Loops



```
for element in iterable:
   body
```



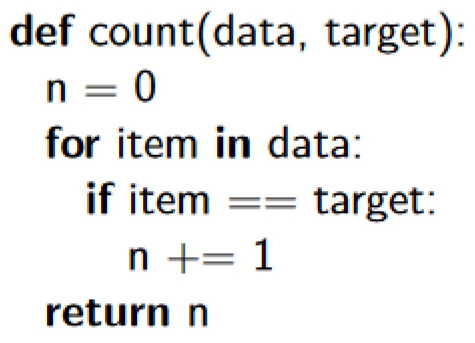
Break and Continue Statements

Python supports a **break** statement that immediately terminate a while or for loop when executed within its body. More formally, if applied within nested control structures, it causes the termination of the most immediately enclosing loop. As a typical example, here is code that determines whether a target value occurs in a data set:

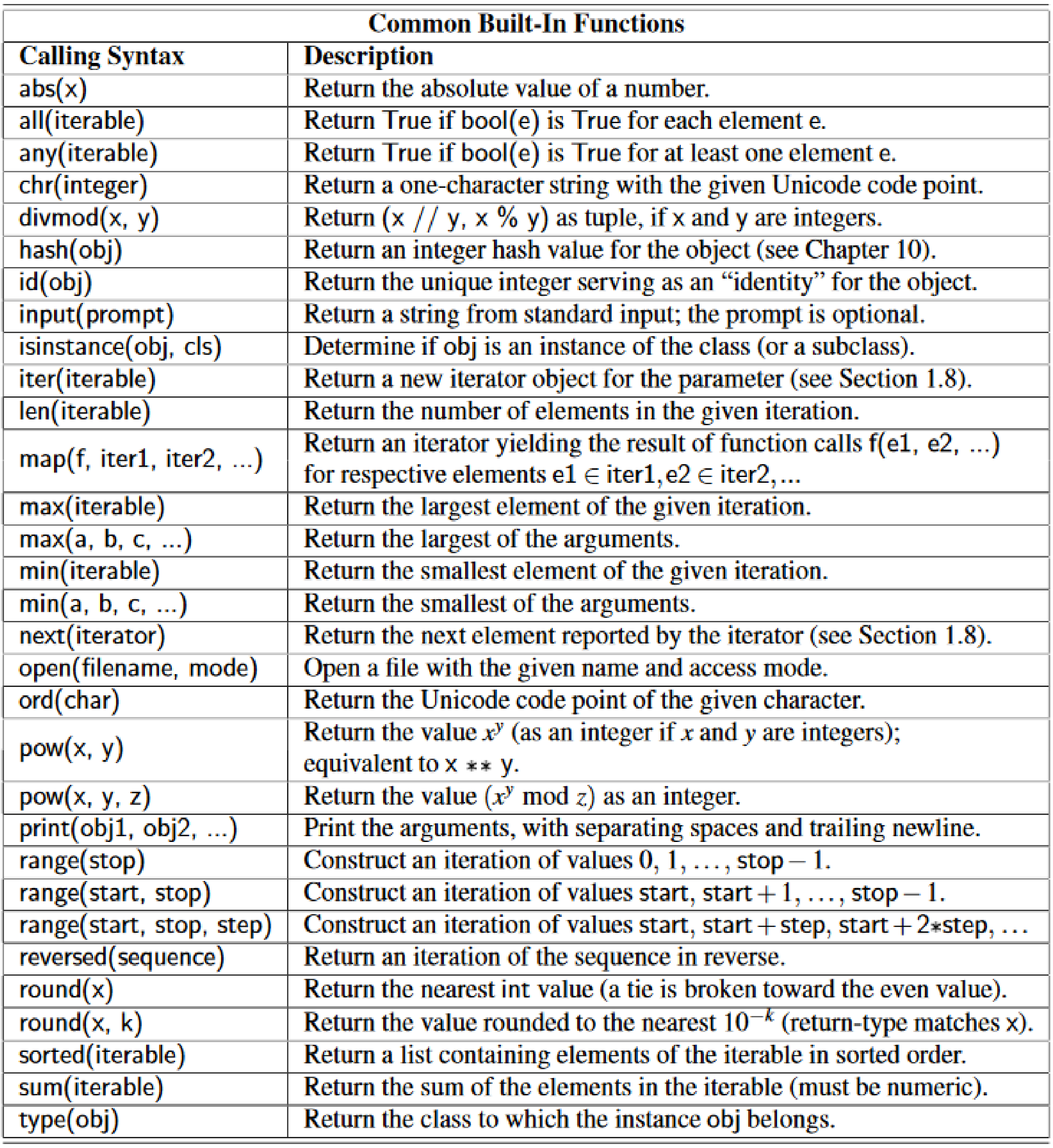
```
found = False
for item in data:
  if item == target:
    found = True
    break
```

Python also supports a **continue** statement that causes the current *iteration* of a loop body to stop, but with subsequent passes of the loop proceeding as expected.

Functions



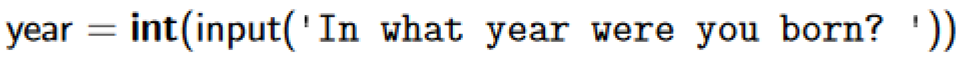
```
def count(data, target):
  n = 0
  for item in data:
    if item == target:
      n += 1
  return n
```



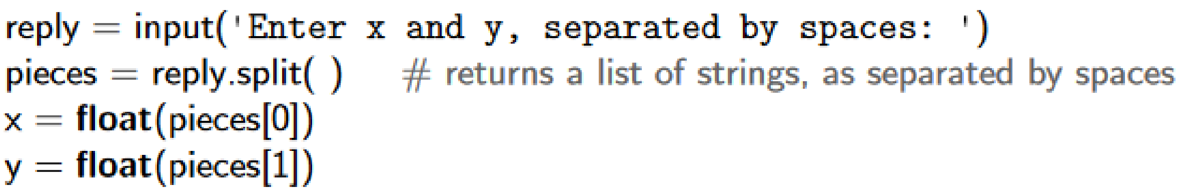
Common Built-In Functions	
Calling Syntax	**Description**
abs(x)	Return the absolute value of a number.
all(iterable)	Return True if bool(e) is True for each element e.
any(iterable)	Return True if bool(e) is True for at least one element e.
chr(integer)	Return a one-character string with the given Unicode code point.
divmod(x, y)	Return (x // y, x % y) as tuple, if x and y are integers.
hash(obj)	Return an integer hash value for the object (see Chapter 10).
id(obj)	Return the unique integer serving as an "identity" for the object.
input(prompt)	Return a string from standard input; the prompt is optional.
isinstance(obj, cls)	Determine if obj is an instance of the class (or a subclass).
iter(iterable)	Return a new iterator object for the parameter (see Section 1.8).
len(iterable)	Return the number of elements in the given iteration.
map(f, iter1, iter2, ...)	Return an iterator yielding the result of function calls f(e1, e2, ...) for respective elements e1 $\in$ iter1, e2 $\in$ iter2, ...
max(iterable)	Return the largest element of the given iteration.
max(a, b, c, ...)	Return the largest of the arguments.
min(iterable)	Return the smallest element of the given iteration.
min(a, b, c, ...)	Return the smallest of the arguments.
next(iterator)	Return the next element reported by the iterator (see Section 1.8).
open(filename, mode)	Open a file with the given name and access mode.
ord(char)	Return the Unicode code point of the given character.
pow(x, y)	Return the value x^y (as an integer if x and y are integers); equivalent to x ** y.
pow(x, y, z)	Return the value (x^y mod z) as an integer.
print(obj1, obj2, ...)	Print the arguments, with separating spaces and trailing newline.
range(stop)	Construct an iteration of values 0, 1, ..., stop − 1.
range(start, stop)	Construct an iteration of values start, start + 1, ..., stop − 1.
range(start, stop, step)	Construct an iteration of values start, start + step, start + 2*step, ...
reversed(sequence)	Return an iteration of the sequence in reverse.
round(x)	Return the nearest int value (a tie is broken toward the even value).
round(x, k)	Return the value rounded to the nearest 10^{-k} (return-type matches x).
sorted(iterable)	Return a list containing elements of the iterable in sorted order.
sum(iterable)	Return the sum of the elements in the iterable (must be numeric).
type(obj)	Return the class to which the instance obj belongs.

Table 1.4: Commonly used built-in function in Python.

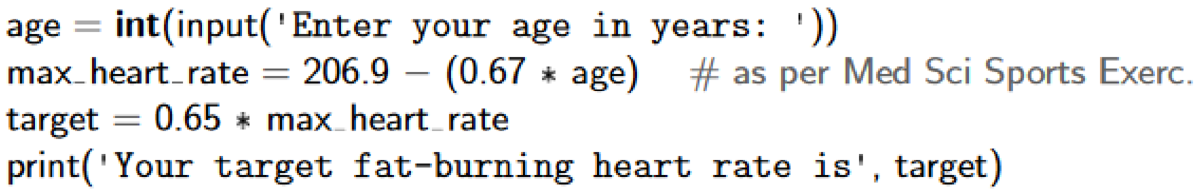
Simple Input and Output



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

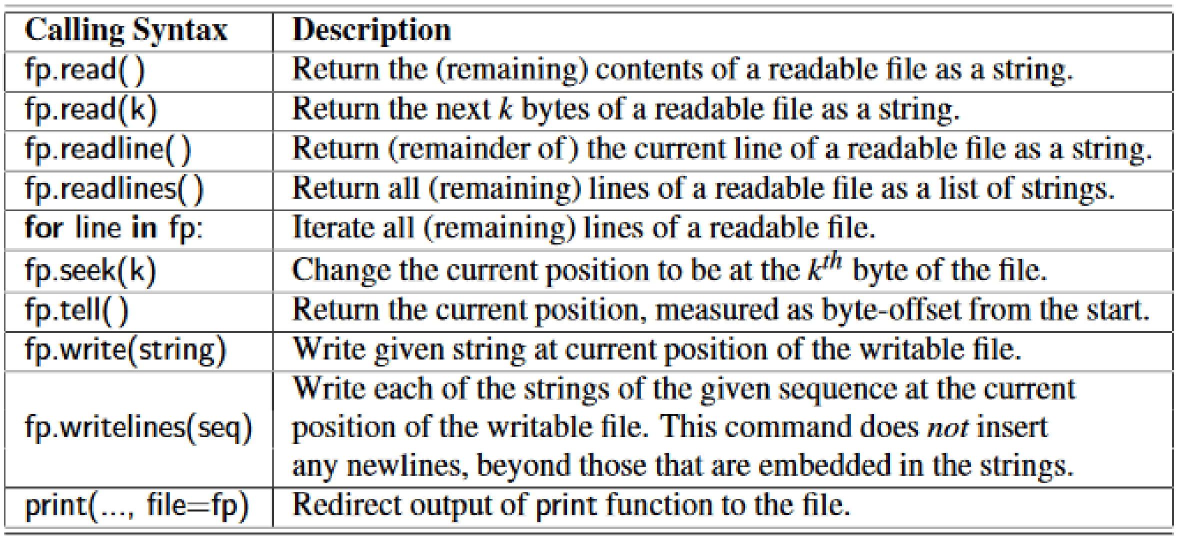


```
reply = input('Enter x and y, separated by spaces: ')
pieces = reply.split( )     # returns a list of strings, as separated by spaces
x = float(pieces[0])
y = float(pieces[1])
```



```
age = int(input('Enter your age in years: '))
max_heart_rate = 206.9 − (0.67 * age)    # as per Med Sci Sports Exerc.
target = 0.65 * max_heart_rate
print('Your target fat-burning heart rate is', target)
```

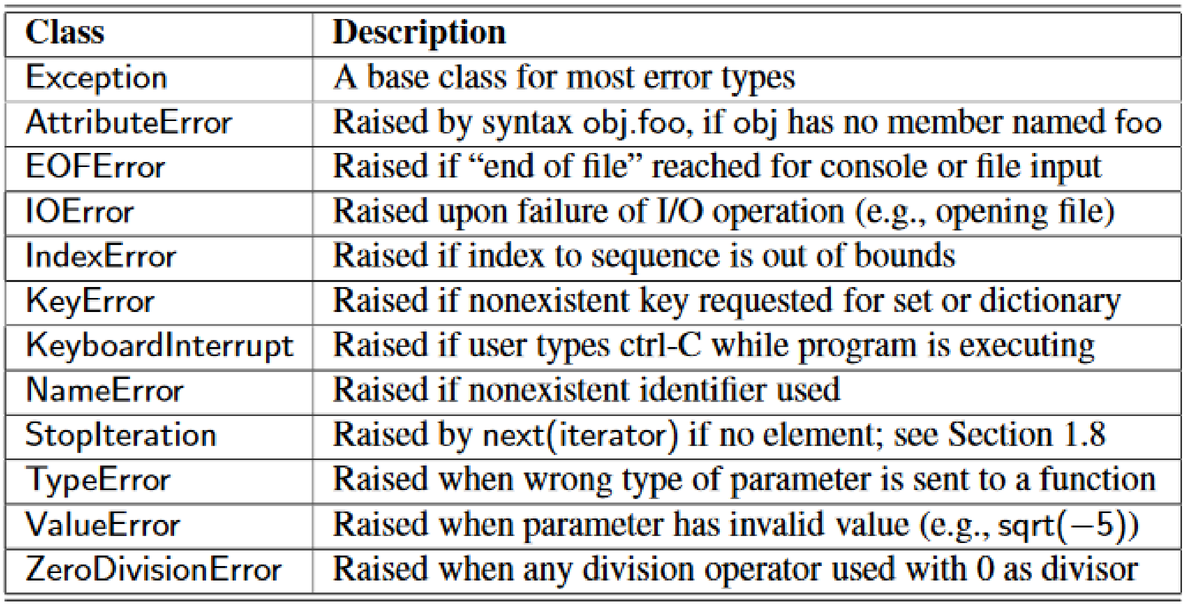
Files



Calling Syntax	Description
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)	Change the current position to be at the k^{th} 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.

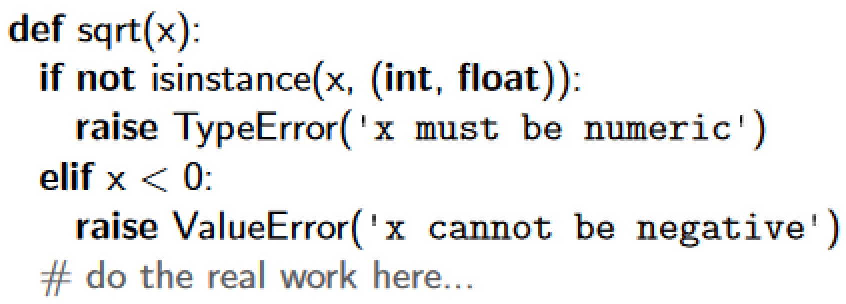
Table 1.5: Behaviors for interacting with a text file via a file proxy (named fp).

Common Exception Types

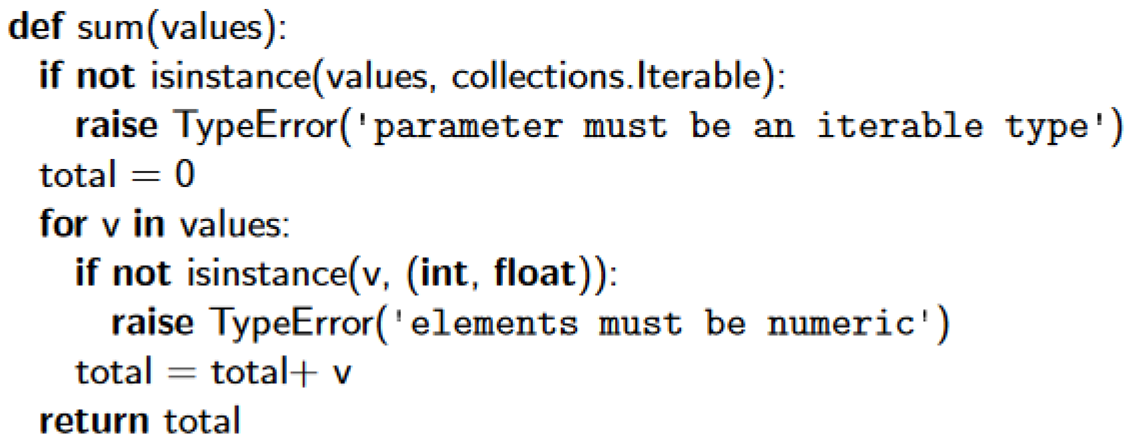


Class	Description
Exception	A base class for most error types
AttributeError	Raised by syntax obj.foo, if obj has no member named foo
EOFError	Raised if "end of file" reached for console or file input
IOError	Raised upon failure of I/O operation (e.g., opening file)
IndexError	Raised if index to sequence 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; see Section 1.8
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

Table 1.6: Common exception classes in Python

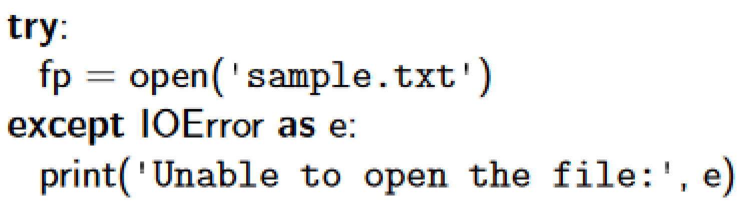


```
def sqrt(x):
  if not isinstance(x, (int, float)):
    raise TypeError('x must be numeric')
  elif x < 0:
    raise ValueError('x cannot be negative')
  # do the real work here...
```

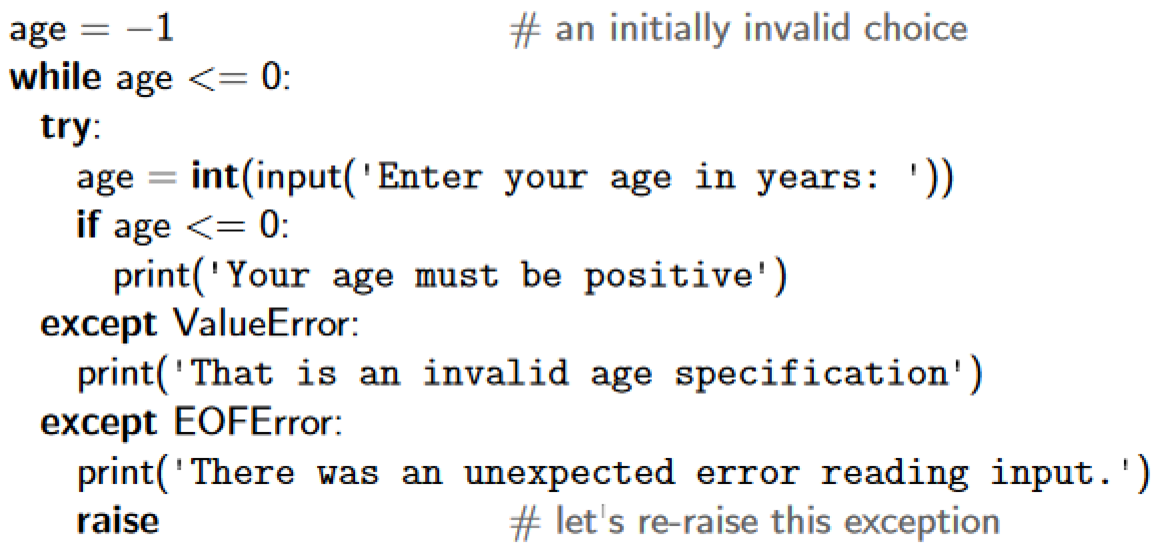


```
def sum(values):
  if not isinstance(values, collections.Iterable):
    raise TypeError('parameter must be an iterable type')
  total = 0
  for v in values:
    if not isinstance(v, (int, float)):
      raise TypeError('elements must be numeric')
    total = total+ v
  return total
```

Exception Handling



```
try:
  fp = open('sample.txt')
except IOError as e:
  print('Unable to open the file:', e)
```



```
age = -1                          # an initially invalid choice
while age <= 0:
  try:
    age = int(input('Enter your age in years: '))
    if age <= 0:
      print('Your age must be positive')
  except ValueError:
    print('That is an invalid age specification')
  except EOFError:
    print('There was an unexpected error reading input.')
    raise                         # let's re-raise this exception
```

```
if n >= 0:
    param = n
else:
    param = −n
result = foo(param)                 # call the function
```

With the conditional expression syntax, we can directly assign a value to variable, param, as follows:

```
param = n if n >= 0 else −n      # pick the appropriate value
result = foo(param)              # call the function
```

As a concrete example, a list of the squares of the numbers from 1 to n, that is $[1,4,9,16,25,\ldots,n^2]$, can be created by traditional means as follows:

```
squares = [ ]
for k in range(1, n+1):
  squares.append(k*k)
```

With list comprehension, this logic is expressed as follows:

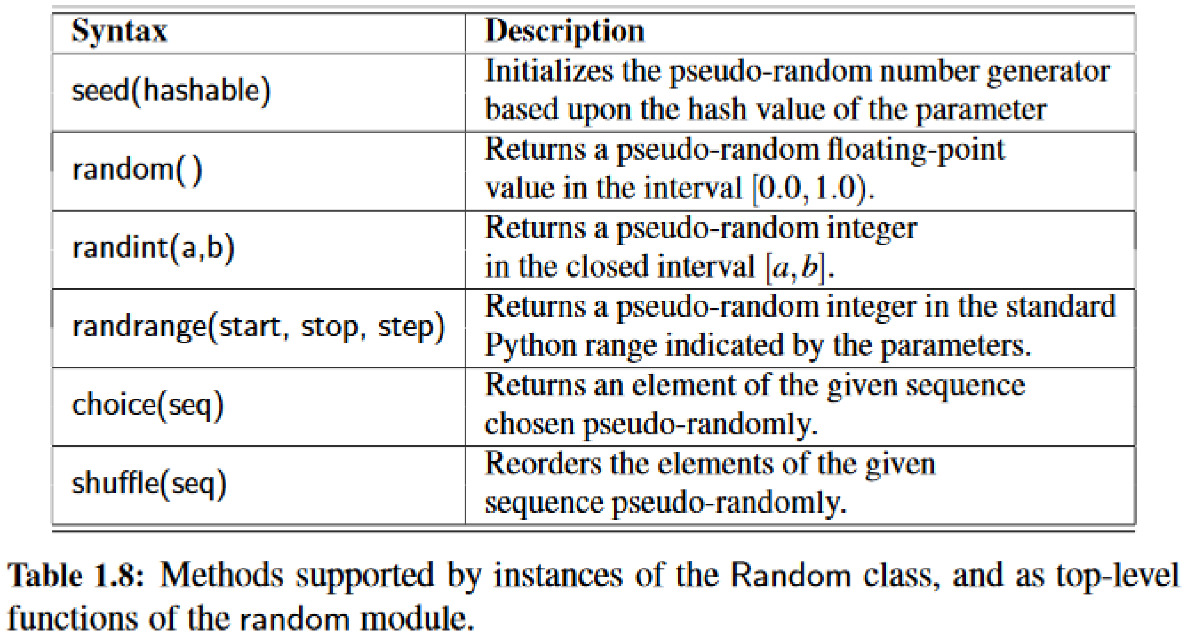
```
squares = [k*k for k in range(1, n+1)]
```

As a second example, Section 1.8 introduced the goal of producing a list of factors for an integer n. That task is accomplished with the following list comprehension:

```
factors = [k for k in range(1,n+1) if n % k == 0]
```

Existing Modules	
Module Name	**Description**
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 (see Section 9.3.7).
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 additional level of interaction with the Python interpreter.
time	Provides support for measuring time, or delaying a program.

Table 1.7: Some existing Python modules relevant to data structures and algorithms.



Syntax	Description
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.

Table 1.8: Methods supported by instances of the Random class, and as top-level functions of the random module.

Python: Most frequently used built-in functions:

- print()

- input()

- type()

- id()

- len()

- int()

- float()

- str()

- bool()

- range()

- filter()

- sorted()

- zip()

- enumerate()

- open()

- max()

- min()

- sum()

- abs()

- round()

- dir()

- help()