01
Python Primer

Chapter 1

Note

This will be just an overview of the language, you are expected to know basic programming concepts and a beginner level knowledge of python.

Python Overview

High-level and object-oriented language

By Guido van Rossum in the early 1990s

Python 2 in 2000

Python 3 in 2008 (what we will use)

www.python.org

The Python Interpreter

Interpreted language

Commands are executed by Python interpreter (takes script as input and reports the results)

Commands can be saved in text file (`demo.py`) and can feed into interpreter as parameter (`python.exe -i demo.py`) or

Interpreter can be used in interactive mode (command-by-command)

Some of the Integrated development environments (IDE) for Python: PyCharm, IDLE

Notepad++ & iPython (a CLI from Anaconda)

White Spaces Matter

Python's syntax relies heavily on the use of white space.

Individual statements concludes with a new line character

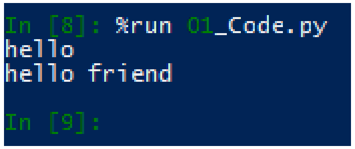
A command can extend to another line

either with a backslash (\) character or

with an open delimiter that has not been closed.

For comments use # character

```
# 01_code.py
print ('hello')

print ('hello\
 friend')

# I'm a comment.

mylist = list([1,2,
3,4])
```



```
In [8]: %run 01_Code.py
hello
hello friend

In [9]:
```

```
def foo():      OK
    return 1

def bar():      IndentationError
return 1
```

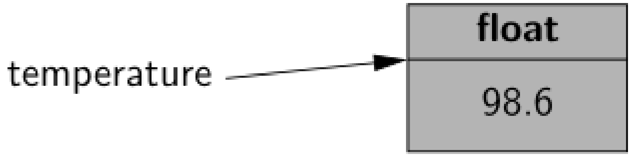
Objects in Python

Classes form the basis for all data types (e.g., int is a class, float is a class, str is class).

```
temperature = 98.6
```

`temperature` is an identifier, `23.2` is an instance of float class.

"The identifier temperature references an instance of the float class."



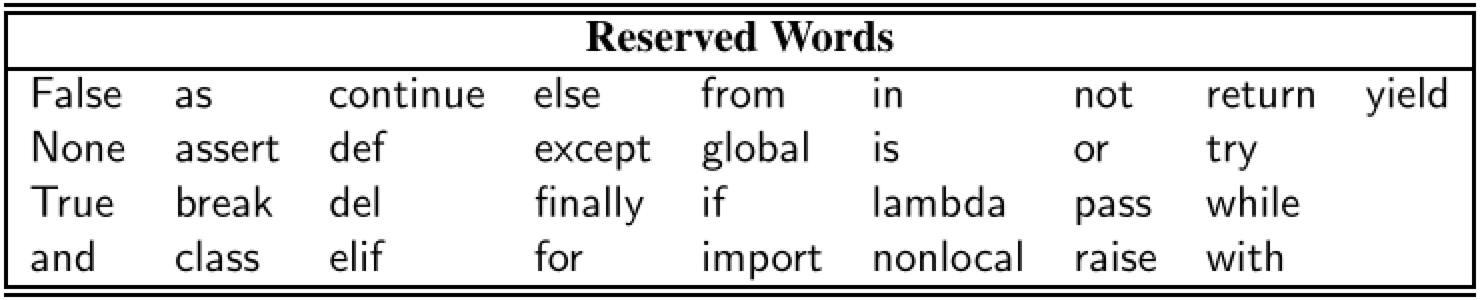
Objects in Python - Identifiers

Identifiers are case-sensitive (temp is not equal to Temp)

They can be composed of any unicode characters

Cannot begin with number (1907bjk is not OK, bjk1903 is OK.)

Identifier cannot be any of the 33 reserved words.



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

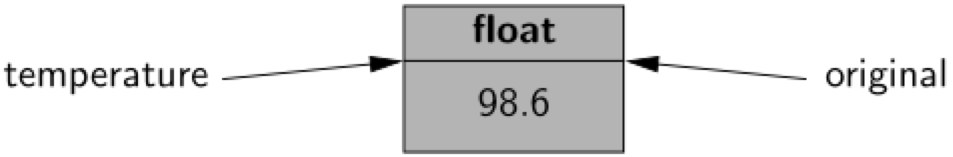
Objects in Python - Identifiers

Identifiers are implicitly associated with the memory address of the object.

Identifiers need not to be declared (unlike in Java or C++)

Identifiers can be associated with any type of object.

An **alias** can be created with `original = temperature`

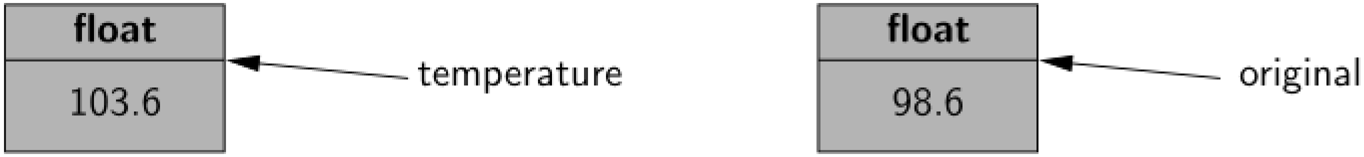


`None` is a special value, identifier is not associated with an object.

Objects in Python - Identifiers

If one of the identifiers is assigned to a new object, that does not affect the aliased object.

```
temperature = temperature + 5.0
```



Objects in Python - Objects

Instantiation: Creating a new object (instance) of class

An object is instatiated by invoking constructor of the class.

A class can have one of more constructors (`Widget()`, `Widget(x,y)`)

Classes have zero or more methods (also a.k.a member functions).

Member functions are called with `.` syntax

```
w = Widget()
w.config(0)
```

Widget object

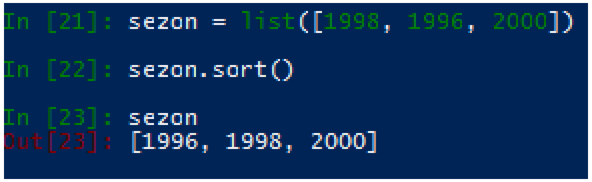
Config object

An member functions can be chained: `w.config(0).reset()`

Objects in Python - Objects

Accessor: Member functions that do not change state of the object (e.g., s = 'kartal'; s.count('a'))

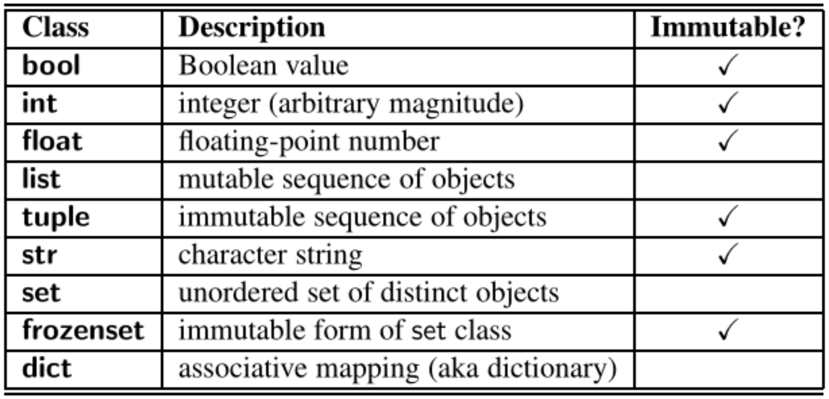
Mutators: Member functions that change the state of the object.



```
In [21]: sezon = list([1998, 1996, 2000])

In [22]: sezon.sort()

In [23]: sezon
Out[23]: [1996, 1998, 2000]
```

Immutable Classes:

Objects of the class cannot be changed after instantiation.



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)	

Objects in Python - Built-in Classes

bool

Logical (Boolean) values

Only two instances exist: `True` and `False`

`bool()` returns `False`

`bool(foo)`

- If foo evaluates to a number, if number is 0 then returns False, otherwise True.
- If foo returns a list or string (or another container type), if the list or string is empty, it returns False, otherwise True.

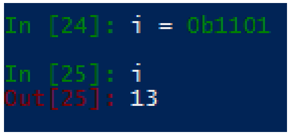
Objects in Python - Built-in Classes

int

Represents integer values with arbitrary magnitude, Python internally manages the representation of an integer.

Typical literals 0, 23, -1303

Integral values can also be expressed as binary, octal, and hexadecimal representations by making use of 0b, 0o, and 0x prefixes, respectively (e.g., 0b1101, 0o73, 0xAB12).

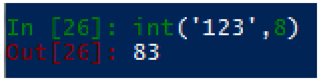


```
In [24]: i = 0b1101

In [25]: i
Out[25]: 13
```

int() returns 0.

int(3.14) evaluates to 3.

int('123') evaluates to 123. int('123', 8) evaluates to 83.



```
In [26]: int('123',8)
Out[26]: 83
```

int('hello') raises ValueError.

Objects in Python - Built-in Classes

float

float is the only floating-point type in Python.

Uses a fixed-precision.

2.0 and 2. are equal.

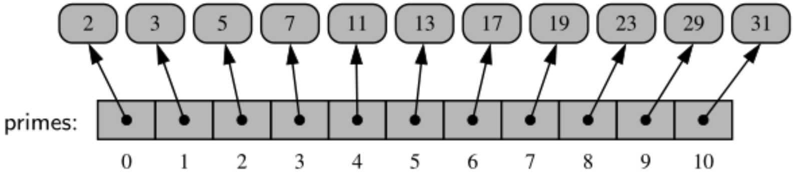
6.022e23 represents 6.022 x 10^{23}

float() returns 0.0

float('3.1') returns 3.1

Objects in Python - Built-in Classes

```
prime = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]
```



Sequence Type: list

Stores a sequence of objects in a referential structure (it stores references to the objects, not the objects themselves).

Array-based sequences, zero-indexed

`[` and `]` are used as delimiters for list literals.

`[1, 2, 3]` is a list containing 3 pointers to different 3 `int` objects.

`list()` produces an empty list.

`list()` can accept any object that is of an iterable type (e.g., list, strings, tuples, sets, dictionaries).

Objects in Python - Built-in Classes

Sequence Type: tuple

An immutable sequence type.

Can be more easily streamlined compared to what can be done with `list`.

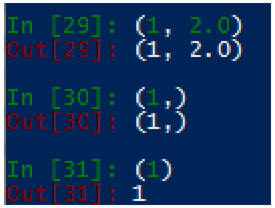
`(` and `)` are used to delimit a tuple.

`()` represents an empty tuple.

`(1 , 2.0)` is tuple with two elements.

`(1,)` is tuple with one element.

`(1)` is not a tuple, it is a statement that evaluates to `1`.



```
In [29]: (1, 2.0)
Out[29]: (1, 2.0)

In [30]: (1,)
Out[30]: (1,)

In [31]: (1)
Out[31]: 1
```

Objects in Python - Built-in Classes

Sequence Type: str

Represents an immutable version unicode character sequence.

> Technically, there exists no character class in Python. Characters are evaluated as strings of length one.

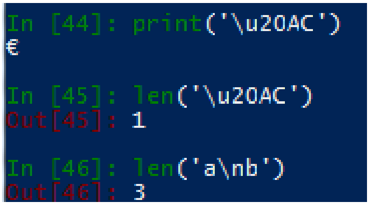
Double quotes or single quote can be used to delimit string literals.

`"I'm OK"` and `'I\'m OK'` represent identical objects.

`\` is the escape character. `\\` backslash, `\t` tab, `\n` new line, `\u` unicode `(\u20AB)`

`"""` or `'''` could be used to delimit string literal.

Improves readability of long strings in code.



```
In [44]: print('\u20AC')
€

In [45]: len('\u20AC')
Out[45]: 1

In [46]: len('a\nb')
Out[46]: 3
```

Objects in Python - Built-in Classes

set and **frozenset**

Set represents the mathematical notion of a set. frozensets are immutable version of sets.

Collection of elements without duplicates.

Elements are not ordered and cannot be ordered.

Based on hash table data structure, they are pretty optimized.

Its elements can only be instances of immutable types (e.g., int, float, str, tuple, frozenset).

{ and } are used to delimit set representations. {12, 'abc'} is a set.

{} is not an empty set but rather it is an empty dictionary.

set() is an empty set.

Objects in Python - Built-in Classes

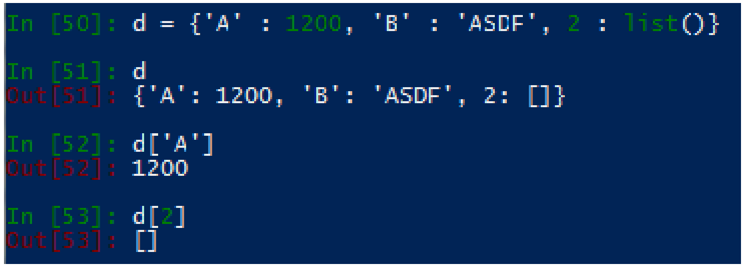
dict

`dict` represents a dictionary (mapping) from a set of distinct keys to associated values.

`{}` is an empty dictionary.

`{'A' : 1200, 'B' : 'ASDF', 2 : list()}` is a dictionary.

Dictionary values can be accessed with an array-like syntax: `d['A']`



```
In [50]: d = {'A' : 1200, 'B' : 'ASDF', 2 : list()}

In [51]: d
Out[51]: {'A': 1200, 'B': 'ASDF', 2: []}

In [52]: d['A']
Out[52]: 1200

In [53]: d[2]
Out[53]: []
```

Expressions, Operators, and Precedence

Logical Operators

`not` unary negation

`and` conditional and

`or` conditional or

`and` and `or` are short-circuit, i.e., in a chained Boolean statement, if the result is determined in the first operation, rest of the calculations are not performed.

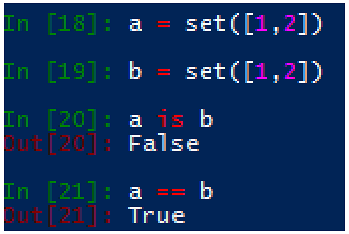
Expressions, Operators, and Precedence

Equality Operators

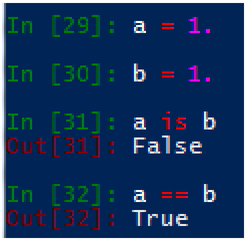
`is`, `is not`, `==`, `!=`

`a is b` evaluates to `True` when they are aliases to the same object.

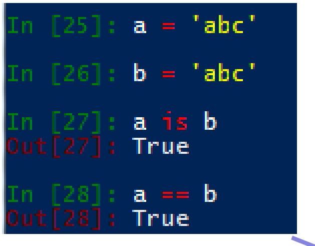
`a == b` evaluates to `True` they are aliases to the same object or when both identifiers refer to different objects that happen to have values considered equal.



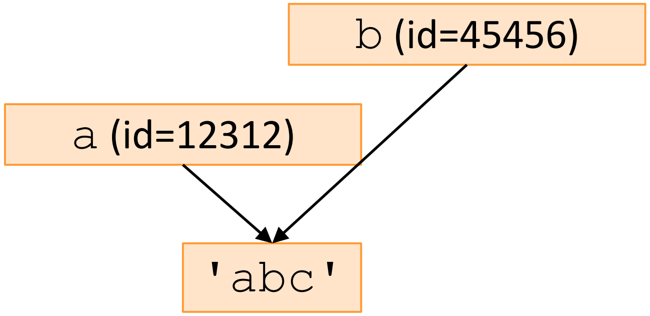
```
In [18]: a = set([1,2])

In [19]: b = set([1,2])

In [20]: a is b
Out[20]: False

In [21]: a == b
Out[21]: True
```



```
In [29]: a = 1.

In [30]: b = 1.

In [31]: a is b
Out[31]: False

In [32]: a == b
Out[32]: True
```



```
In [25]: a = 'abc'

In [26]: b = 'abc'

In [27]: a is b
Out[27]: True

In [28]: a == b
Out[28]: True
```



b (id=45456)

a (id=12312)

'abc'

string interning

Expressions, Operators, and Precedence

Comparison Operators

<, <=, >, >=

In number types, these operators behave as expected.

In strings, they compare lexicographic order case-sensitively.

Expressions, Operators, and Precedence

Arithmetic Operators

+, -, * addition, subtraction, multiplication

/ true division (5/2 = 2.5)

// integer division (5//2 = 2)

% the modulo operator (5 % 4 = 1)

% operator works also with negative and floating point numbers. Fro further details, see textbook.

Expressions, Operators, and Precedence

Bitwise Operators

~ bitwise complement (prefix unary operator)

& bitwise and

| bitwise or

ˆ bitwise exclusive-or

<< shift bits left, filling in with zeros

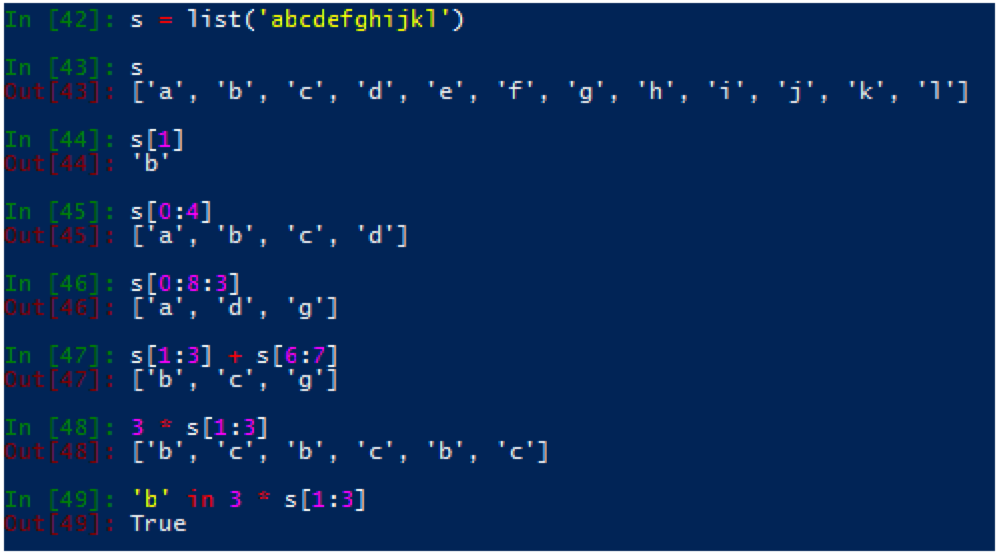
>> shift bits right, filling in with sign bit

Expressions, Operators, and Precedence

Sequence Operators

Sequence types (`str`, `tuple`, and `list`) support the following operators.

`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



```
In [42]: s = list('abcdefghijkl')

In [43]: s
Out[43]: ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l']

In [44]: s[1]
Out[44]: 'b'

In [45]: s[0:4]
Out[45]: ['a', 'b', 'c', 'd']

In [46]: s[0:8:3]
Out[46]: ['a', 'd', 'g']

In [47]: s[1:3] + s[6:7]
Out[47]: ['b', 'c', 'g']

In [48]: 3 * s[1:3]
Out[48]: ['b', 'c', 'b', 'c', 'b', 'c']

In [49]: 'b' in 3 * s[1:3]
Out[49]: True
```

Expressions, Operators, and Precedence

Sequence Operators

`s[-1]` denotes last element in the sequence.

`s[-2]` denotes second last element in the sequence.

`del s[1]` drops the designated element from the sequence.

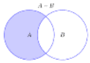
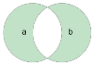
in can also used for substring check (`'acik' in 'arabacik'`)

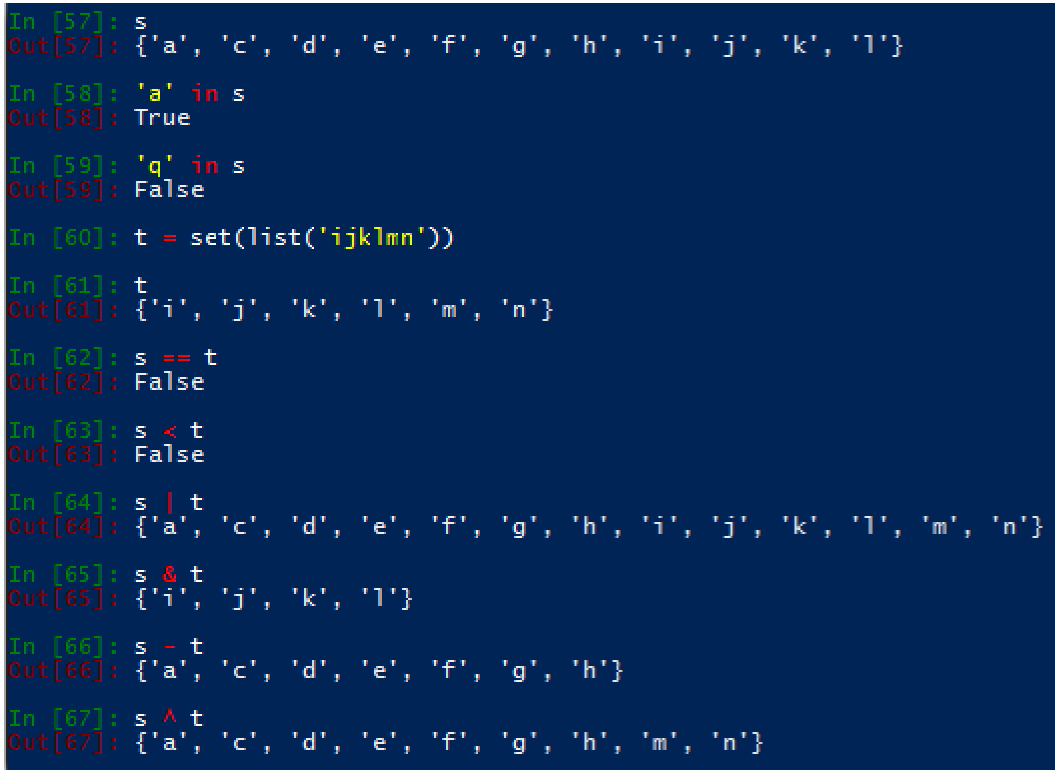
Sequences can also be compared according to lexicographic order:
`[5, 6, 9] < [5, 7]` evaluates to `True`.

`s == t` equivalent (elemen by element)

Expressions, Operators, and Precedence

Set 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



```
In [57]: s
Out[57]: {'a', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l'}

In [58]: 'a' in s
Out[58]: True

In [59]: 'q' in s
Out[59]: False

In [60]: t = set(list('ijklmn'))

In [61]: t
Out[61]: {'i', 'j', 'k', 'l', 'm', 'n'}

In [62]: s == t
Out[62]: False

In [63]: s < t
Out[63]: False

In [64]: s | t
Out[64]: {'a', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n'}

In [65]: s & t
Out[65]: {'i', 'j', 'k', 'l'}

In [66]: s - t
Out[66]: {'a', 'c', 'd', 'e', 'f', 'g', 'h'}

In [67]: s ^ t
Out[67]: {'a', 'c', 'd', 'e', 'f', 'g', 'h', 'm', 'n'}
```

Expressions, Operators, and Precedence

Dictionary Operators

`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

Expressions, Operators, and Precedence

Extended Assignment Operators

`+=` and `-=` operators are also supported for certain data types.

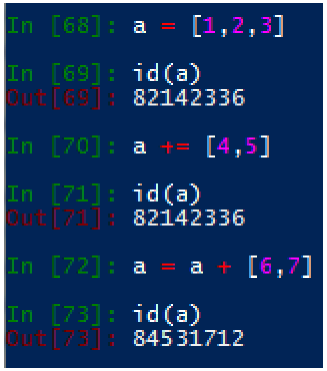
However, they should be used carefully.

For example, for `list`s, expressions

`a = a + [4, 5]` and `a += [4, 5]` have same result, however,

`a += [4, 5]` mutates existing object whereas

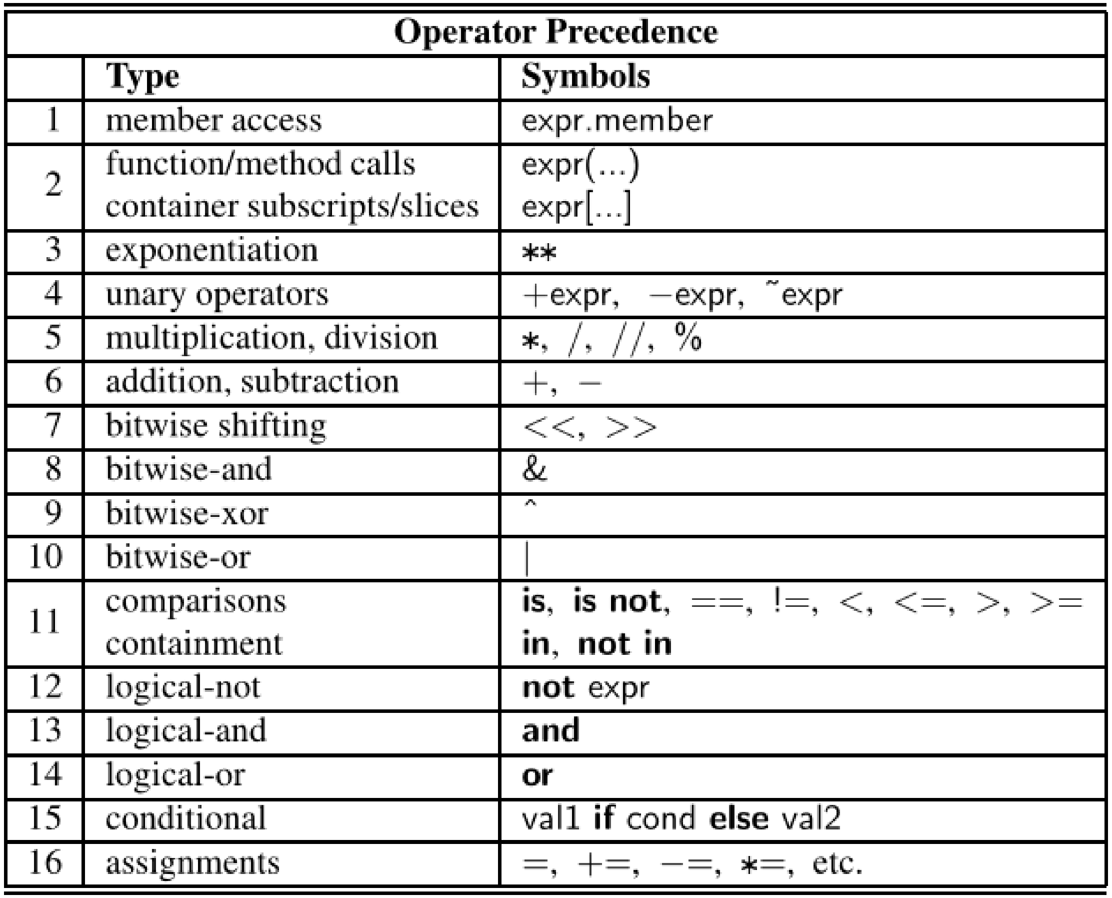
`a = a + [4, 5]` generate new object and makes a new assignment.



```
In [68]: a = [1,2,3]

In [69]: id(a)
Out[69]: 82142336

In [70]: a += [4,5]

In [71]: id(a)
Out[71]: 82142336

In [72]: a = a + [6,7]

In [73]: id(a)
Out[73]: 84531712
```

Expressions, Operators, and Precedence

Operator Precedence



	Operator Precedence	
	Type	**Symbols**
1	member access	expr.member
2	function/method calls container subscripts/slices	expr(...) 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 containment	**is**, **is not**, ==, !=, <, <=, >, >= **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.

Chain assignment is also supported:

x = y = 0

Chaining of comparison operators:

1 <= x + y <= 10

is equal to

(1 <= x + y) and (x + y <= 10)

Control Flow

if-elif-else

if statements provide a way to execute a chosen block of code.

elif and else statements are optional.

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

Control Flow

while

Body code block is executed as long as condition is evaluated to True.

```
while condition:
    body
```

Also note that if condition is a string or number, it will be cast with bool() constructor. For example,

```
x = 2
while x: # Think as if "while bool(x):"
  x = x - 1
  print(x)
```

will yield

```
1
0
```

and stop.

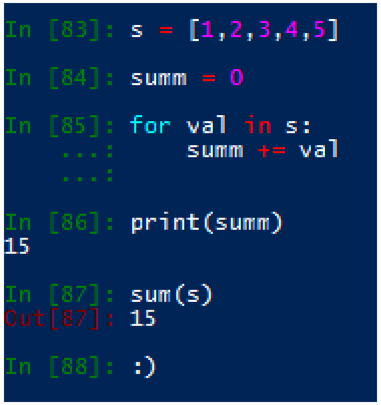
Control Flow

for

iterable can be any iterable structure (e.g., `list`, `tuple`, `set`, `dict`, `file`).

Syntax is analogous to `foreach` in Java.

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



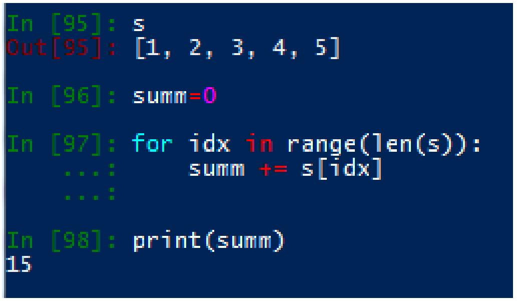
```
In [83]: s = [1,2,3,4,5]

In [84]: summ = 0

In [85]: for val in s:
    ...:     summ += val
    ...:

In [86]: print(summ)
15

In [87]: sum(s)
Out[87]: 15

In [88]: :)
```

Control Flow

for (index-based)

If we would like to access each item by an index value, then we would need to create a series with `range`.

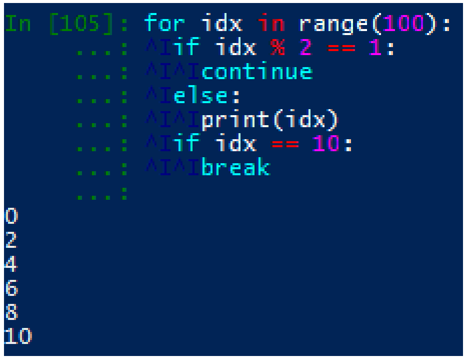
```
for element in iterable:
    body
```



```
In [95]: s
Out[95]: [1, 2, 3, 4, 5]

In [96]: summ=0

In [97]: for idx in range(len(s)):
    ...:     summ += s[idx]
    ...:

In [98]: print(summ)
15
```

`break` breaks the loop,

`continue` skips the current iteration of a loop.



```
In [105]: for idx in range(100):
     ...: ^Iif idx % 2 == 1:
     ...: ^I^Icontinue
     ...: ^Ielse:
     ...: ^I^Iprint(idx)
     ...: ^Iif idx == 10:
     ...: ^I^Ibreak
     ...:
0
2
4
6
8
10
```

Functions

```
def print_increment(num):   # signature of function
    incnum = num + 1        # body
    print(incnum)           # body
    return incnum           # body
```

In run time, a new identifier with the name of the function (e.g., `print_increment`) is created.

When the function is called, an activation record is created.

Activation record holds:

(1) A namespace for the function's local scope

(2) Under this namespace, function's parameters are defined, and

(3) Any other local identifiers are registered to the name space.

Functions

```
def print_increment(num):   # signature of function
    incnum = num + 1        # body
    print(incnum)           # body
    return incnum           # body
```

`return` statement quits the execution of the function.

If it has a parameter, function returns the value of that parameter to the caller.

If no parameter is specified or function execution reaches to the end of the body of the function without executing any `return` statement, `None` is returned to the caller.

Functions

formal parameter

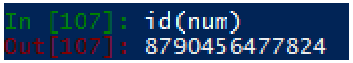
```
def print_increment(num):   # signature of function
     incnum = num + 1        #
    print(incnum)           #
    return incnum           # body
```

actual parameter

What happens when the call `print_increment(inputnum)` is performed?

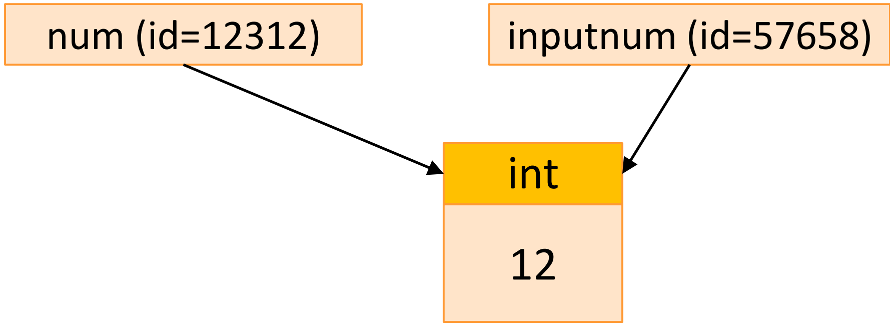
`def print_increment(num):`
The signature is checked, a new identifier is created with name of the formal parameter.

num (id=12312)



```
In [107]: id(num)
Out[107]: 8790456477824
```

The new identifier is assigned to the "object" which is identified by `inputnum`. Technically, a new additional alias is created for the object passed as parameter.



"Formal" parameter is an alias to the "actual" parameter.

Functions

formal parameter

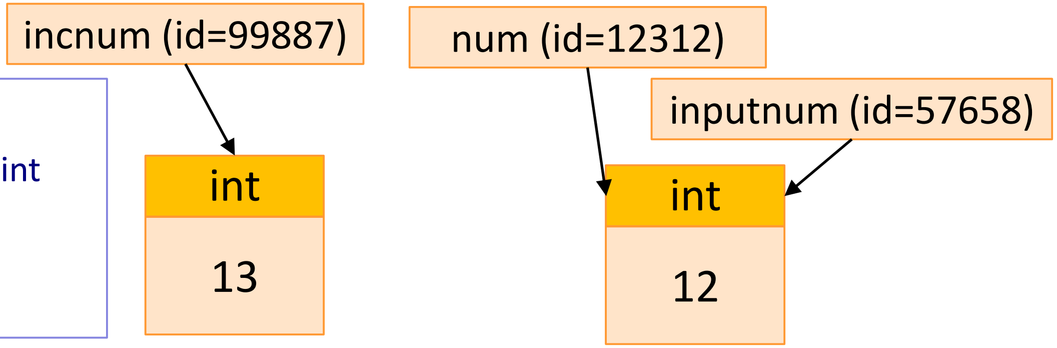
```
def print_increment(num):   # signature of function
    incnum = num + 1        #
    print(incnum)           #
    return incnum           # body
```

actual parameter

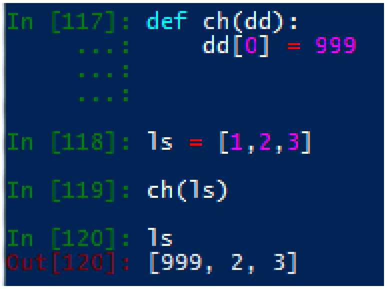
What happens when the call `print_increment(inputnum)` is performed?

```
incnum = num + 1
```

A new object is created by incrementing the int object which is the input parameter. And, an identifier, namely incnum, is assigned to this newly created object.



Please note that, the actual parameter value did not change. However if we were to have a mutable input parameter such as list, the actual parameter would change after the execution of the function.



```
In [117]: def ch(dd):
     ...:     dd[0] = 999
     ...:
     ...:

In [118]: ls = [1,2,3]

In [119]: ch(ls)

In [120]: ls
Out[120]: [999, 2, 3]
```

Functions

Functions are said to be **polymorphic** if they support more than one calling signature.

Such functions' signatures have parameter(s) with default values.

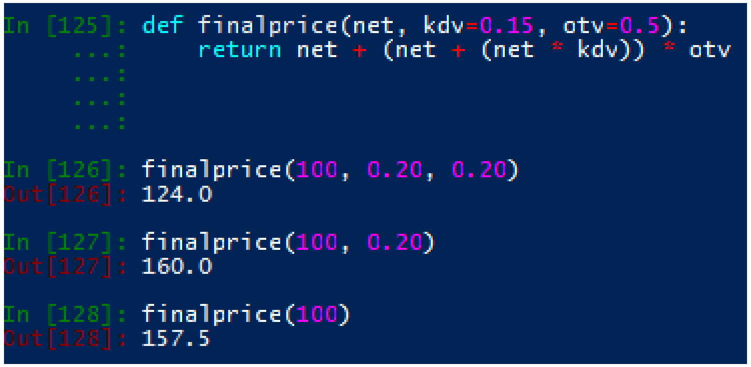
```
def foo(a, b=12, c=24):
```

All acceptable calls:

```
foo(5)
foo(5, 3)
foo(5, 3, 2)
```

positional arguments
formal parameters are assigned to actual parameters according to their order

Recall that the function range had three different invocation options: `range(10)`, `range(0,10)`, `range(0,10,2)`



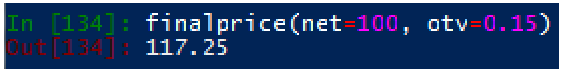
```
In [125]: def finalprice(net, kdv=0.15, otv=0.5):
     ...:     return net + (net + (net * kdv)) * otv
     ...:
     ...:
     ...:

In [126]: finalprice(100, 0.20, 0.20)
Out[126]: 124.0

In [127]: finalprice(100, 0.20)
Out[127]: 160.0

In [128]: finalprice(100)
Out[128]: 157.5
```

Functions

Keyword Arguments

In keyword argument mechanism, actual parameter is explicitly assigned to implicit parameter.

For example:



```
In [134]: finalprice(net=100, otv=0.15)
Out[134]: 117.25
```

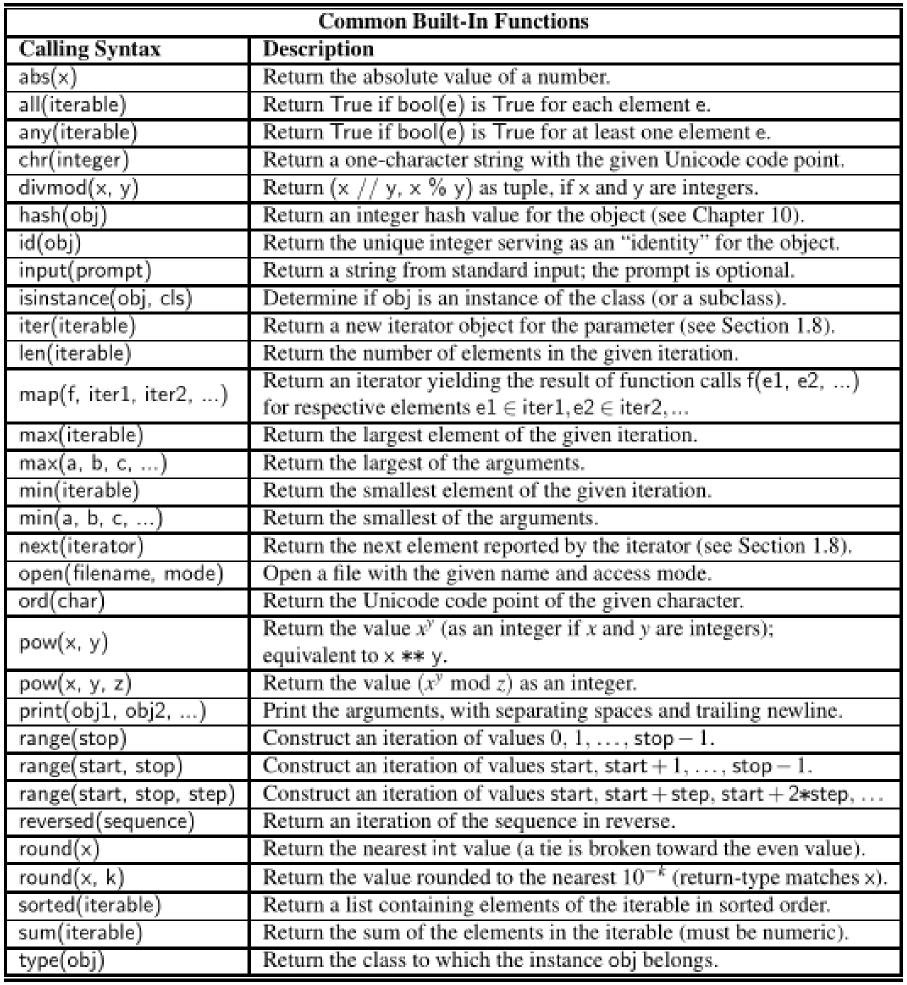
`foo(c=5)`

Keyword arguments should follow positional arguments.

`print(1, 2, 3, sep=',')` is allowed.

`print(1, sep=',', 2, 3)` is not allowed.

Built-in Functions



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.

Notice the nuances of different signatures of same functions (e.g. `max`).

For the full list, see here: https://docs.python.org/3/library/functions.html

Input/Output

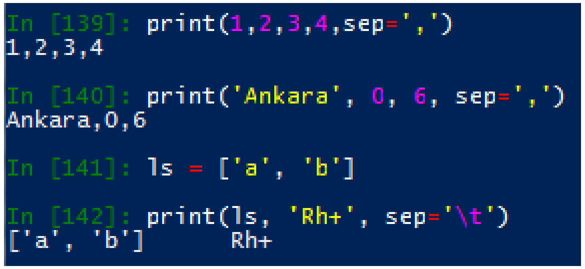
print

Writes its params to a `file` (default: "standard output.")

By separating them with `sep` (e.g., `\t` or `'  ')` (default: space)

Followed by `end` (e.g., `\n`). (default: new line)

It can take as many positional arguments as possible.



```
In [139]: print(1,2,3,4,sep=',')
1,2,3,4

In [140]: print('Ankara', 0, 6, sep=',')
Ankara,0,6

In [141]: ls = ['a', 'b']

In [142]: print(ls, 'Rh+', sep='\t')
['a', 'b']      Rh+
```

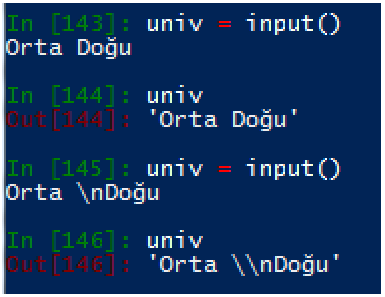
Input/Output

input

With the input function, user input on the console can be acquired.

User input is return as a string object.

Any sequence of characters, except for the new line character, are returned to the caller.



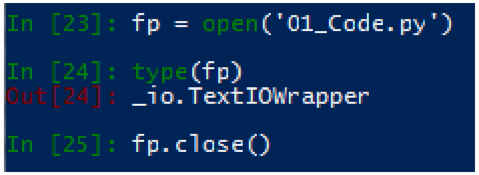
```
In [143]: univ = input()
Orta Doğu

In [144]: univ
Out[144]: 'Orta Doğu'

In [145]: univ = input()
Orta \nDoğu

In [146]: univ
Out[146]: 'Orta \\nDoğu'
```

Files

open

Returns a handle to the file whose name is specified as an argument.

Files can be opened with different modes: `'r'`, `'w'`, `'a'`, `'wb'`, `'rb'`, etc.



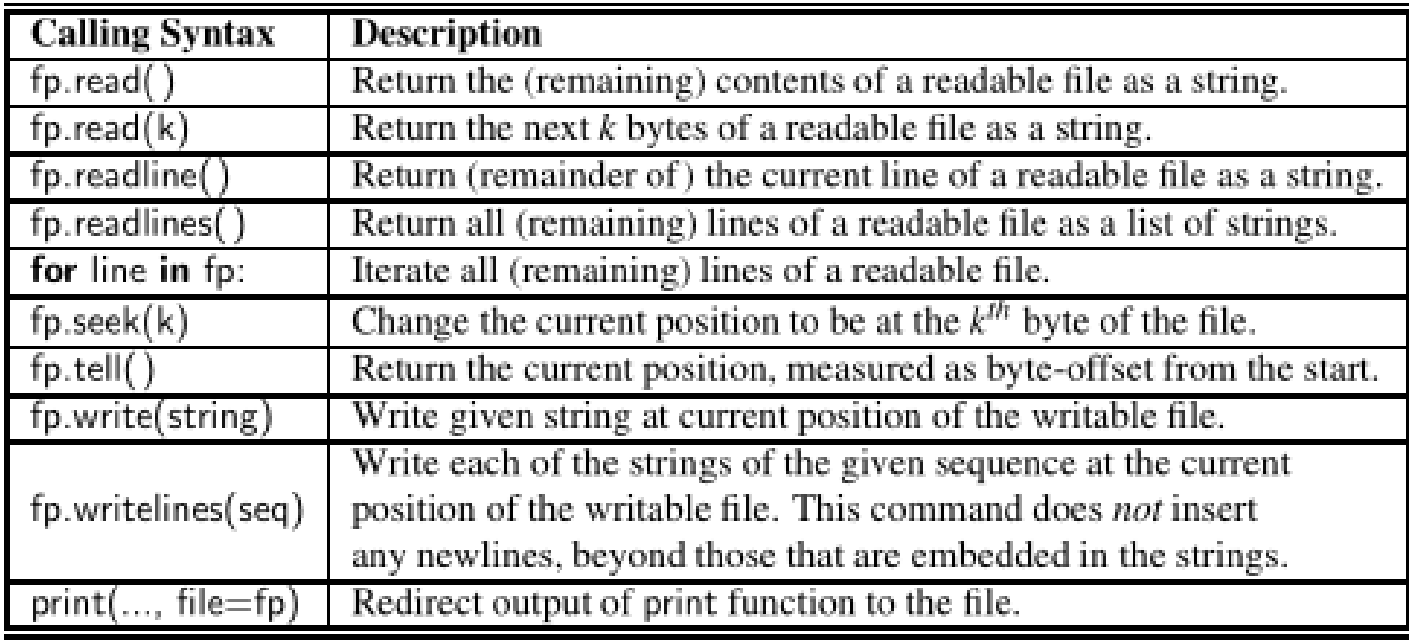
```
In [23]: fp = open('01_Code.py')

In [24]: type(fp)
Out[24]: _io.TextIOWrapper

In [25]: fp.close()
```

close

Closes the file and makes sure any changes are persisted to disk.

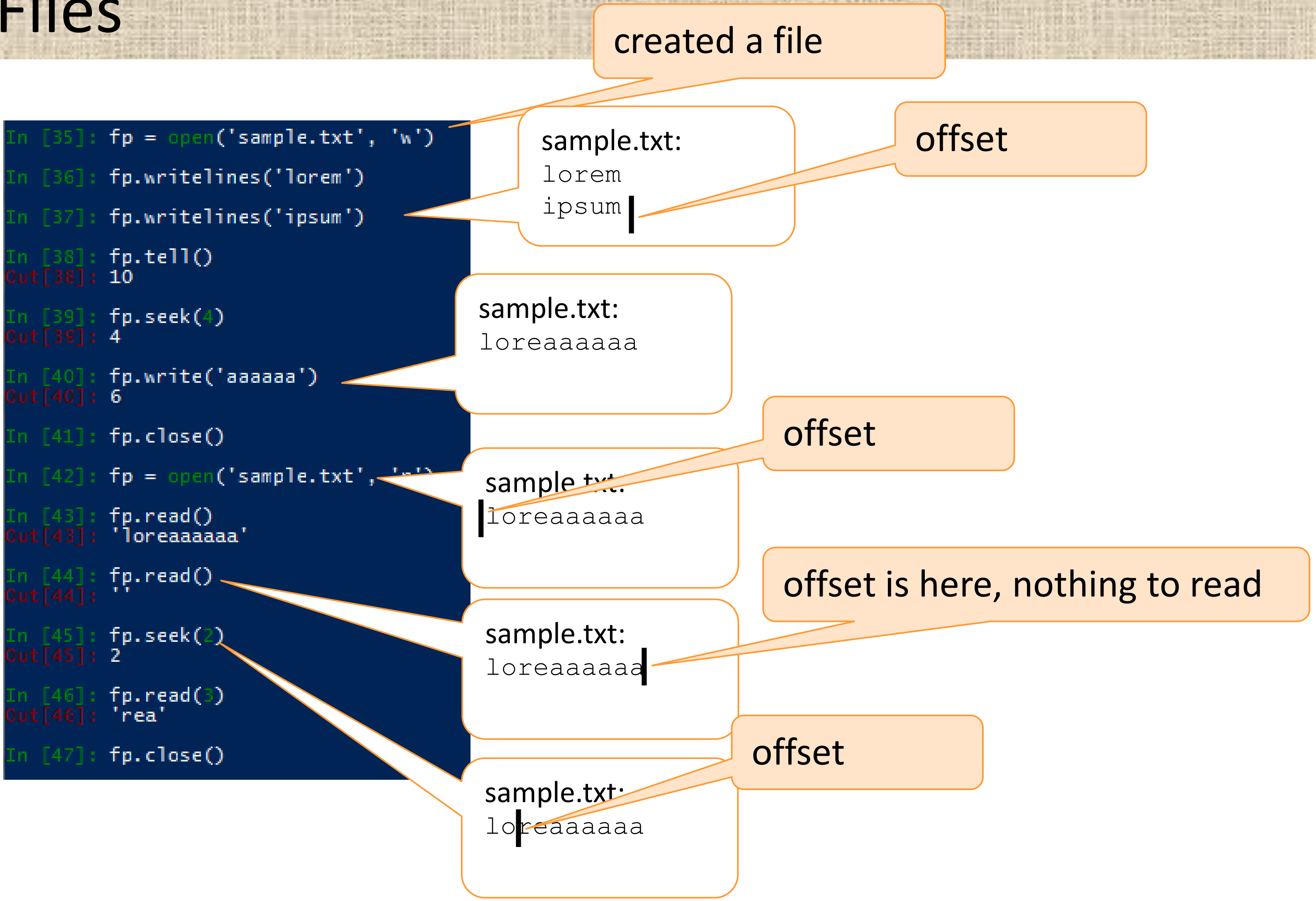
Files

Other File/File Handler Methods



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.

Files



```
In [35]: fp = open('sample.txt', 'w')

In [36]: fp.writelines('lorem')

In [37]: fp.writelines('ipsum')

In [38]: fp.tell()
Out[38]: 10

In [39]: fp.seek(4)
Out[39]: 4

In [40]: fp.write('aaaaaa')
Out[40]: 6

In [41]: fp.close()

In [42]: fp = open('sample.txt', 'r')

In [43]: fp.read()
Out[43]: 'loreaaaaaa'

In [44]: fp.read()
Out[44]: ''

In [45]: fp.seek(2)
Out[45]: 2

In [46]: fp.read(3)
Out[46]: 'rea'

In [47]: fp.close()
```

created a file

sample.txt:
```
lorem
ipsum
```

offset

sample.txt:
```
loreaaaaaa
```

offset

sample.txt:
```
loreaaaaaa
```

offset is here, nothing to read

sample.txt:
```
loreaaaaaa
```

offset

sample.txt:
```
loreaaaaaa
```

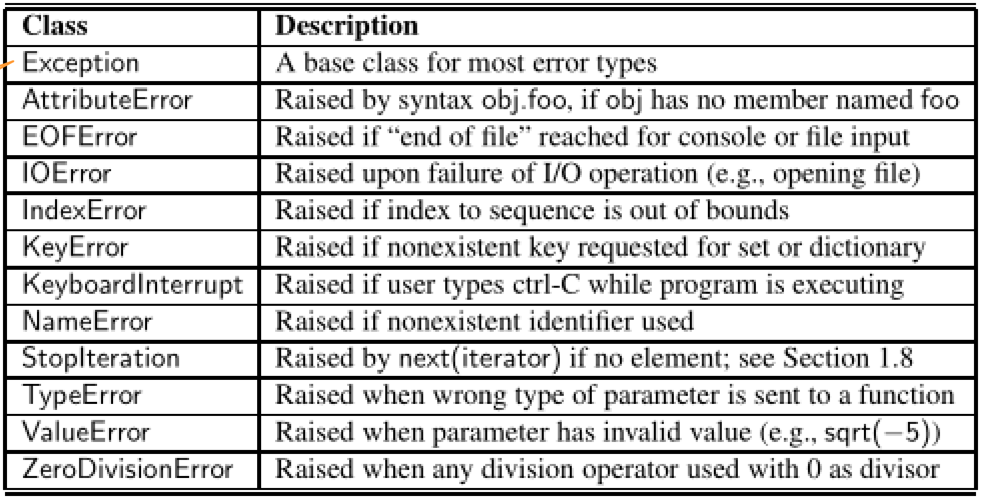
Exception Handling

In coding, errors occur.

Code raises exceptions when it enters to an unexpected state:

- Being out of memory
- Null reference
- I/O error, etc.

Base class for the rest of the 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

Exception Handling

Exceptions are raised with `raise` function.

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

An example of useful case: Parameter checking

```
def calc_average_grade(val_list):
    for val in val_list:
        if not isinstance(val ,(float, int)):
            raise TypeError('val has to be of type int or float')
        if val < 0 or val > 100:
            raise ValueError('val has to be in range [0-100]')
    # do stuff
```

Exception Handling

We can handle selected exceptions in Python by making use of try-except block.

```
try:
    # ...
    try:
        .
        .
        fp = open( sample.txt )
        .
        .
    except IOError as e:
        print('Unable to open the file:', e)
    # ...
    # ...
except ValueError('...') as f:
        print ('Some value error occurred.')
```

Iterators and Generators

Iterator: An object that can manage an iteration through a series of variables.

Iterable Object: An object that can produce an iterator.

```
data = [1,2,3] # an iterable object
i = iter(data) # an iterator
next(i) # returns next element in the iterable object
next(data) # error!
```

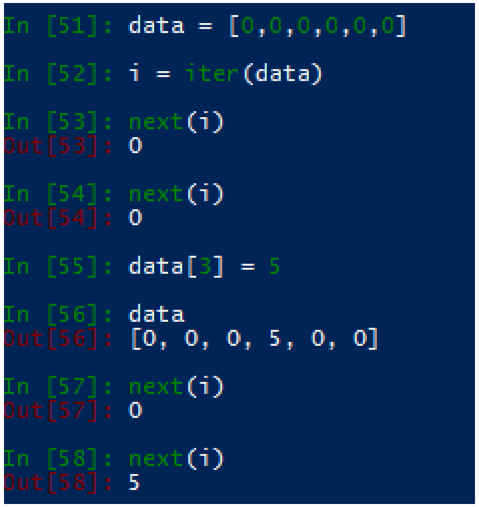
`list`, `tuple`, `set`, `string`, `dict`, `file`, user-defined objects, etc.

Iterators and Generators

Iterators does not maintain a copy of the iterable object.

They work on an "index" (a pointer) to the items in the iterable object.

If the iterable items change, the iterator will point to the updated items.

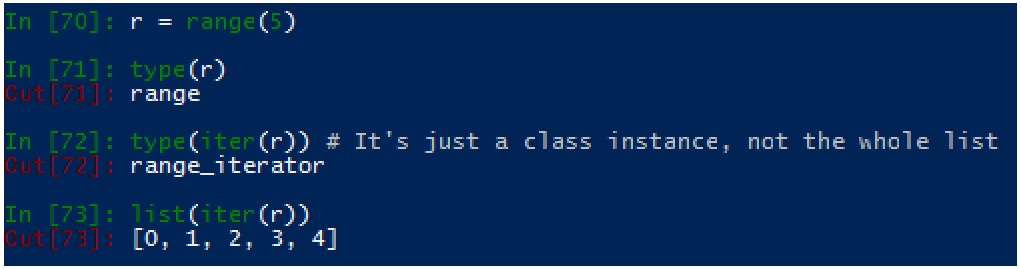


```
In [51]: data = [0,0,0,0,0,0]

In [52]: i = iter(data)

In [53]: next(i)
Out[53]: 0

In [54]: next(i)
Out[54]: 0

In [55]: data[3] = 5

In [56]: data
Out[56]: [0, 0, 0, 5, 0, 0]

In [57]: next(i)
Out[57]: 0

In [58]: next(i)
Out[58]: 5
```

Iterators and Generators

In many of the Python libraries, items of iterable objects are **lazily evaluated**: Items are generated as needed, one at a time.

`range(1000)` does not generate 1000 items. It generates an iterable object which can produce results (items) as needed.



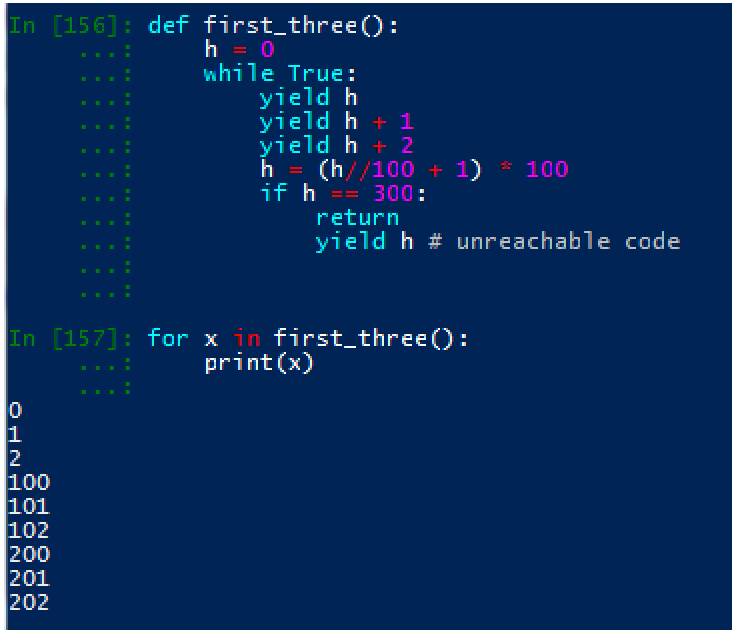
```
In [70]: r = range(5)

In [71]: type(r)
Out[71]: range

In [72]: type(iter(r)) # It's just a class instance, not the whole list
Out[72]: range_iterator

In [73]: list(iter(r))
Out[73]: [0, 1, 2, 3, 4]
```

Iterators and Generators

Generators

Function-like stuctures generating an element of a series logically defined in the implementation.

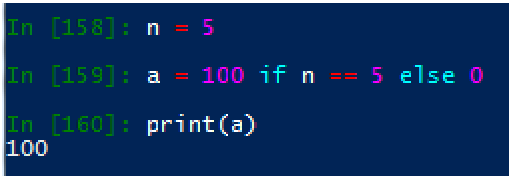
- `yield` statement necessary
- may have zero-argument `return` statements.
- Execution is paused whenever a `yield` statement is reached.
- Execution is resumed whenever next value is requested until the next `yield` statement or a `return` statement.



```
In [156]: def first_three():
     ...:     h = 0
     ...:     while True:
     ...:         yield h
     ...:         yield h + 1
     ...:         yield h + 2
     ...:         h = (h//100 + 1) * 100
     ...:         if h == 300:
     ...:             return
     ...:             yield h # unreachable code
     ...:
     ...:

In [157]: for x in first_three():
     ...:     print(x)
     ...:
0
1
2
100
101
102
200
201
202
```

Syntax Conveniences

`expr1 if condition else expr2`



```
In [158]: n = 5

In [159]: a = 100 if n == 5 else 0

In [160]: print(a)
100
```

Comprehension Syntax

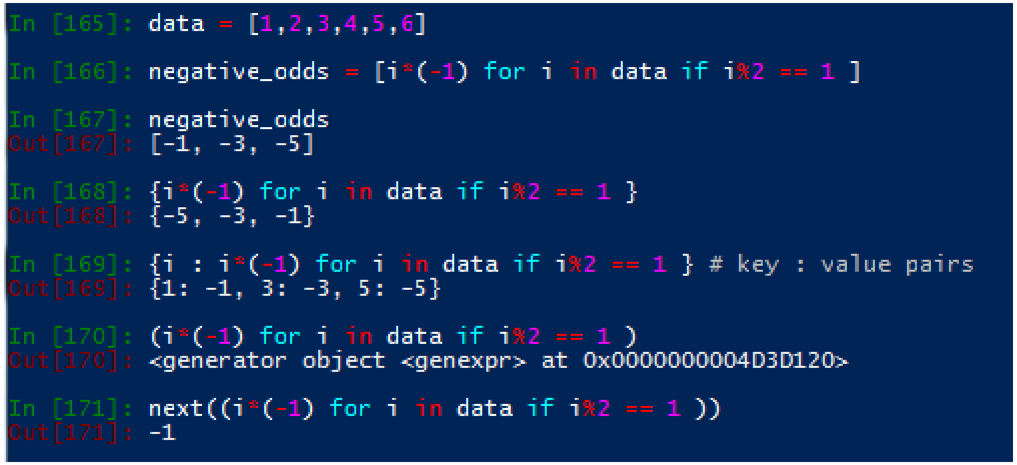
`[ expression for value in iterable if condition ]`

`list` comprehension

`set` comprehension

`dict` conprehension

`generator` comprehension



```
In [165]: data = [1,2,3,4,5,6]

In [166]: negative_odds = [i*(-1) for i in data if i%2 == 1 ]

In [167]: negative_odds
Out[167]: [-1, -3, -5]

In [168]: {i*(-1) for i in data if i%2 == 1 }
Out[168]: {-5, -3, -1}

In [169]: {i : i*(-1) for i in data if i%2 == 1 } # key : value pairs
Out[169]: {1: -1, 3: -3, 5: -5}

In [170]: (i*(-1) for i in data if i%2 == 1 )
Out[170]: <generator object <genexpr> at 0x0000000004D3D120>

In [171]: next((i*(-1) for i in data if i%2 == 1 ))
Out[171]: -1
```

Syntax Conveniences

Automatic tuple packing

Comma-separated values are automatically evaluted as tuple.

Quite helpful when returning multiple values from a function.

```
data = 1, 2, 3 # (1,2,3) a tuple object
```

```
def dummy:
    return 1, 2 # returns (1,2), i.e., a tuple object
```

Unpacking

```
a, b = dummy() # a = 1, b = 2
for x, y in [ (7, 2), (5, 8), (6, 4) ]:
```

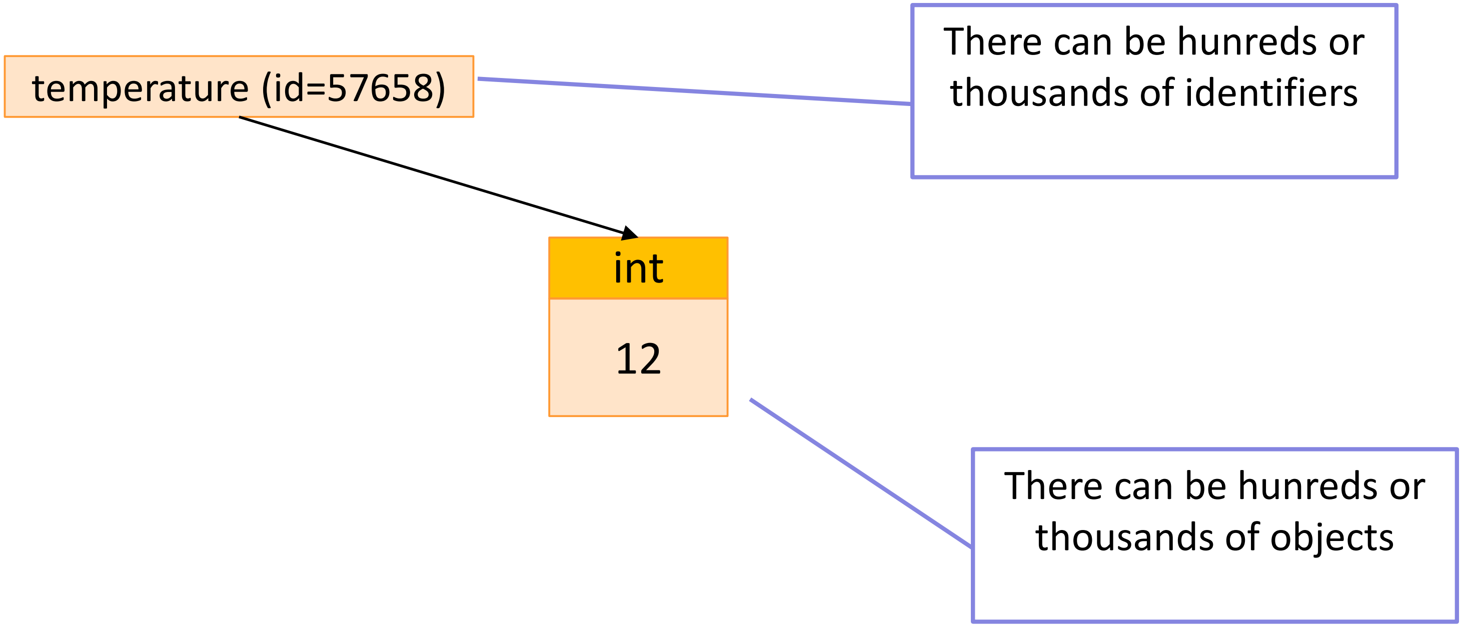
Syntax Conveniences

Simultaneous Assignment

```
x, y, z = 6, 2, 5
j, k = k, j
# instead of
temp = j
j = k
k = temp
```

Scopes and Namespaces

```
temperature = 12
```



temperature (id=57658)

int

12

There can be hunreds or thousands of identifiers

There can be hunreds or thousands of objects

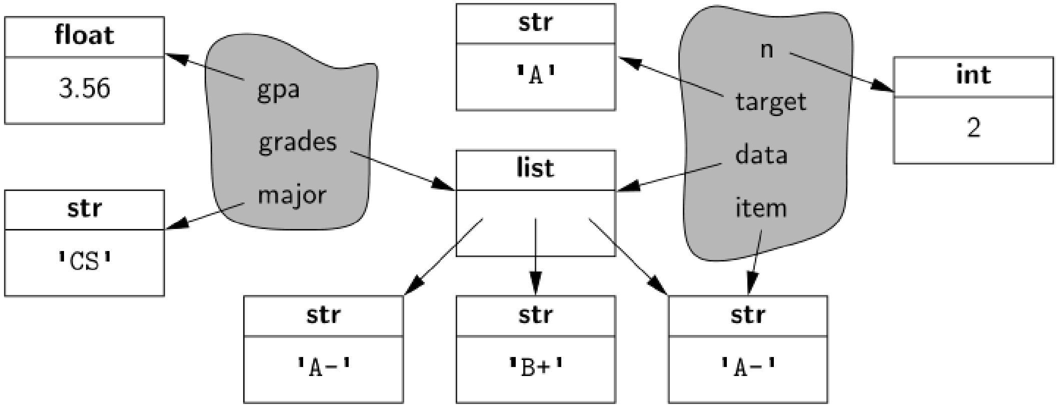
Scopes and Namespaces

Name Resolution: Determining the value associated with an identifier. (What value does identifier x has?)

Scope: Defines in which level the identifier was assigned to a value. *"A scope is a textual region of a Python program where a namespace is directly* [without using fully qualified reference] *accessible."*

Namespace: A collection of identifiers defined in a given scope.

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



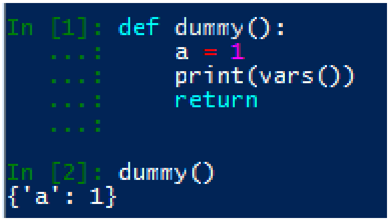
Scopes and Namespaces

Python maintains a **dictionary** in which identifiers and values are stored.

`dir()` reports the names of the identifiers (i.e., keys)

`vars()` reports names and values (i.e., key-value pairs)

`dir` and `vars` operate on the most locally enclosing namespace.



```
In [1]: def dummy():
   ...:     a = 1
   ...:     print(vars())
   ...:     return
   ...:

In [2]: dummy()
{'a': 1}
```

`dir(__builtins__)` lists the identifiers in the builtins namespace.

Technically, `__builtins__` is the identifier of `builtins` module, and, by default, exists in the **global namespace**. When a module object is given as a parameter to `dir`, it lists all the identifiers introduced in that module.

Scopes and Namespaces

Python searches for an identifier in the namespaces in the following order (**LEGB**):

1. Local namespace (e.g., inside a function)
2. Encapsulating namespace(s) (e.g., inside a class or module)
3. Global namespace (i.e., names at the level of main program)
4. Built-ins namespace (exists and available throughout the execution)

Modules

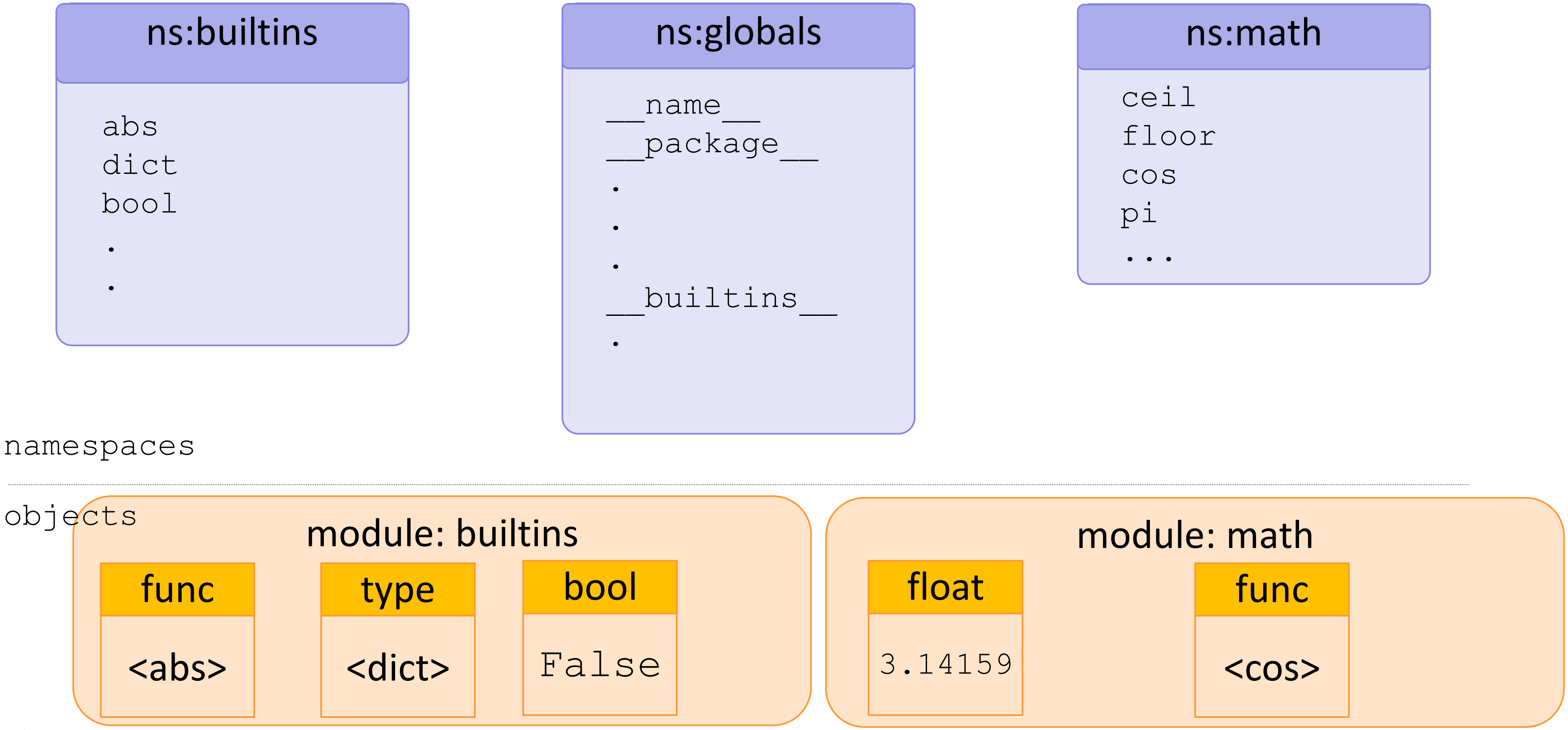
Modules are libraries, containers for values, functions, and classes that are logically related.

Modules have namespaces.

Python's built-in functions reside in builtins module, whose name space is `builtins`.

`builtins` is the default namespace and exists during the execution of Python program.

Modules



ns:builtins

abs
dict
bool
.
.

ns:globals

__name__
__package__
.
.
.
__builtins__
.

ns:math

ceil
floor
cos
pi
...

namespaces

objects

module: builtins

func	type	bool
<abs>	<dict>	False

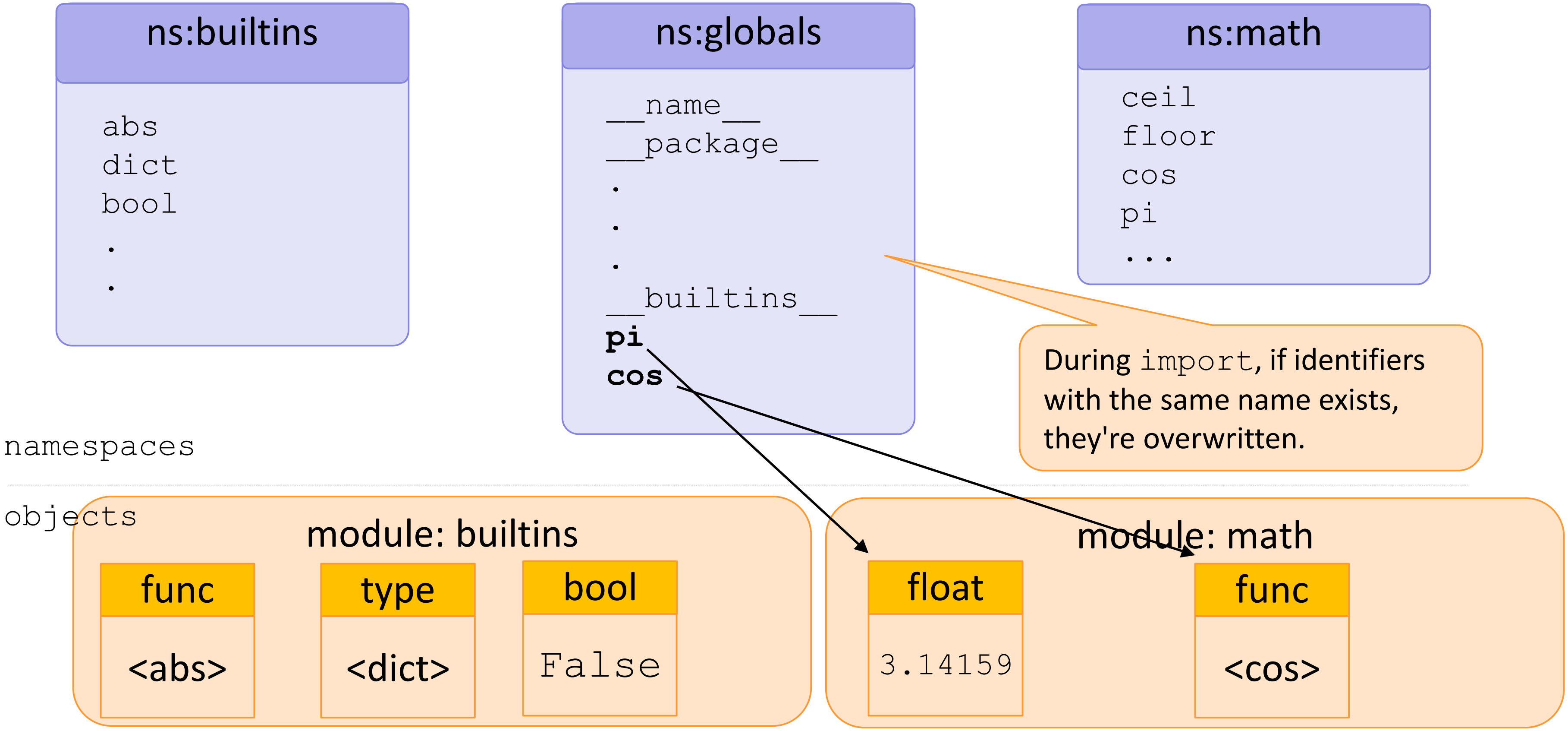
module: math

float	func
3.14159	<cos>

For functions and constants residing in built-in namespace, check these links:
https://docs.python.org/3/library/functions.html#built-in-funcs
https://docs.python.org/3/library/constants.html#built-in-consts

Modules

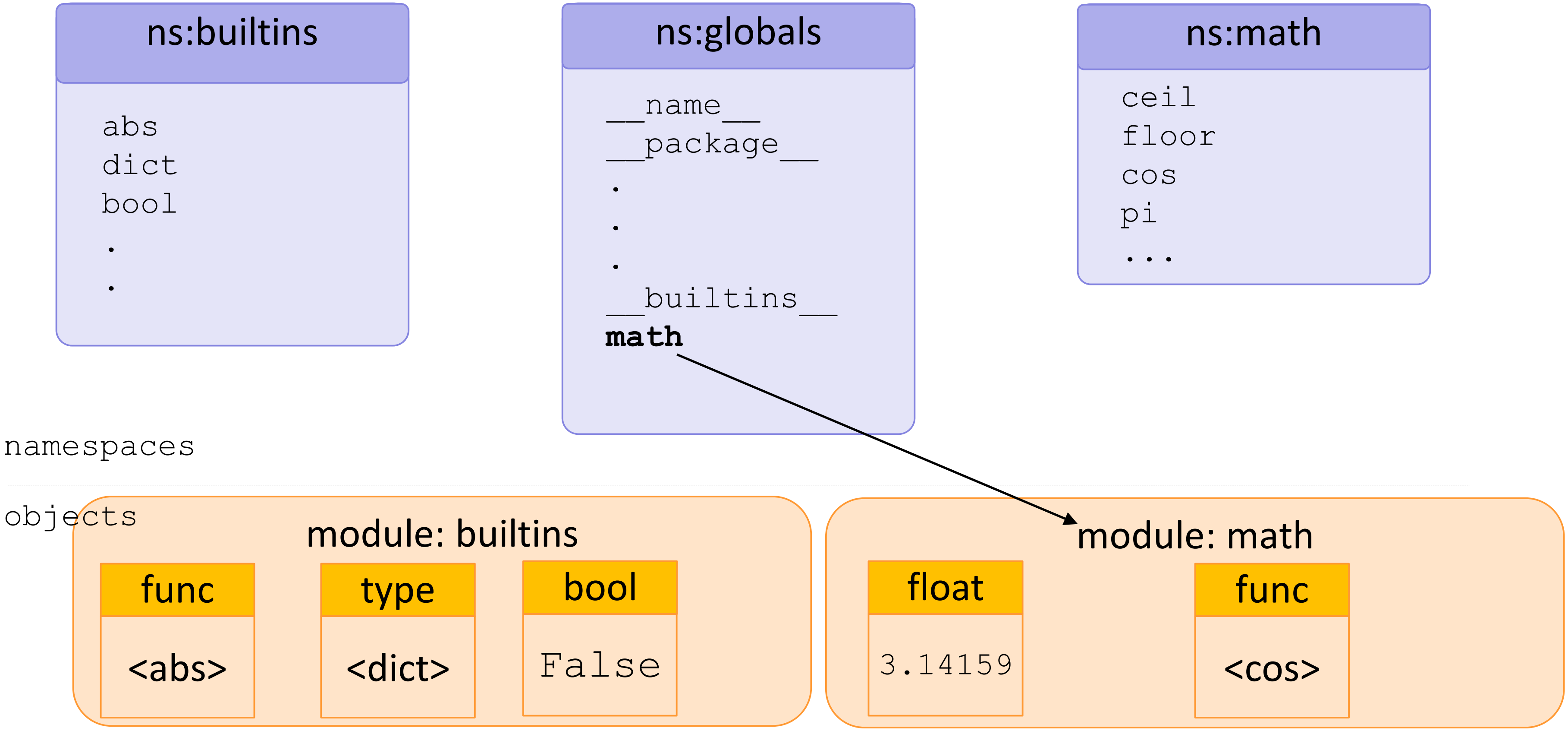
```
from math import pi, cos
```



During import, if identifiers with the same name exists, they're overwritten.

Modules

```
import math
```



Modules

User-defined modules can be created by simply gathering relevant definitions in a file having `.py` extension (e.g., `<module_name>.py`).

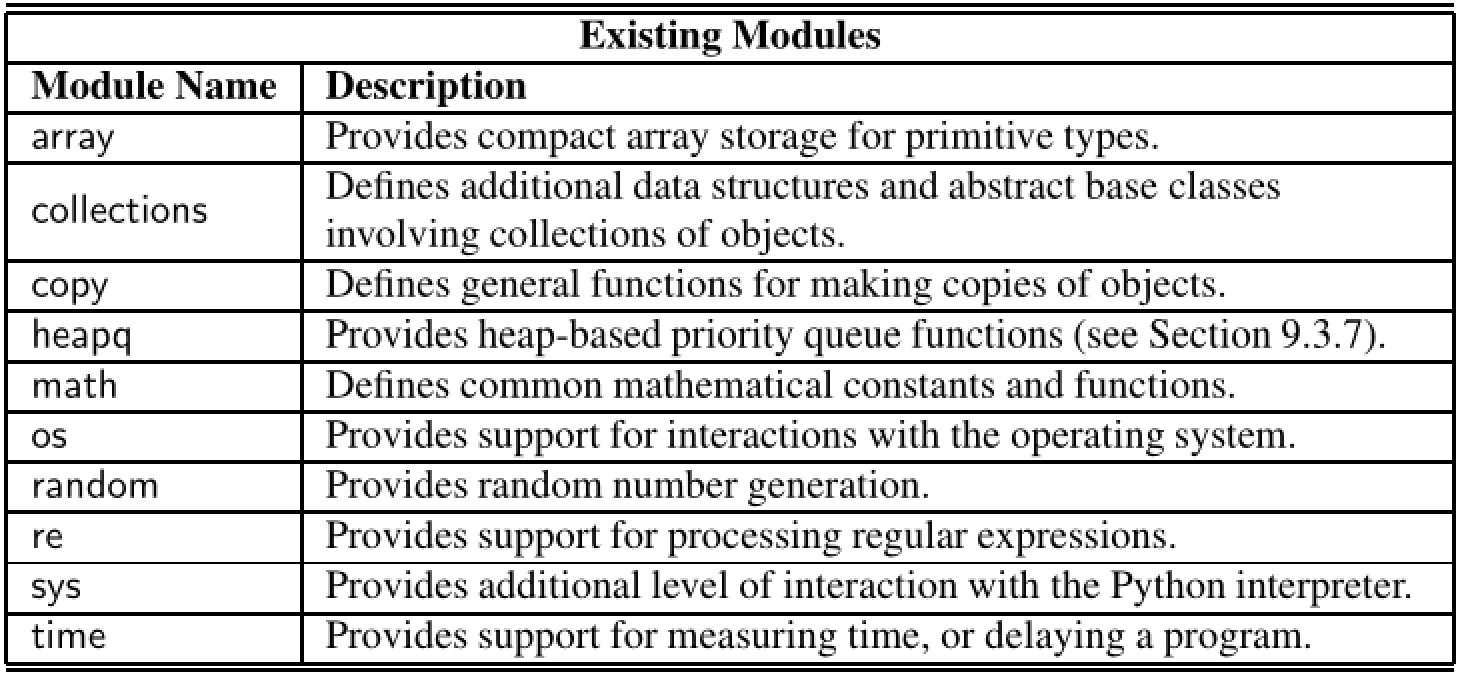
Each of the script in the file is invoked when the module is imported for the first time.

In order to run the module as a standalone program (i.e., $> `python.exe <module_name>.py`), the following construct should be added to module:

```
if name == __main__ :
```

Modules

Existing modules relevant to data structures and algorithms.



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.