Introducing Python Object Types

In an informal sense, in Python we do things with stuff.1 “Things” take the form of operations like addition and concatenation, and “stuff” refers to the objects on which we perform those operations. In this part of the book, our focus is on that stuff, and the things our programs can do with it. Somewhat more formally, in Python, data takes the form of objects—either built-in objects that Python provides, or objects we create using Python classes or external language tools such as C extension libraries. Although we’ll firm up this definition later, objects are essentially just pieces of memory, with values and sets of associated operations. As we’ll see, everything is an object in a Python script

From a more concrete perspective, Python programs can be decomposed into modules, statements, expressions, and objects, as follows:

1. Programs are composed of modules.
2. Modules contain statements.
3. Statements contain expressions.
4. Expressions create and process objects

We will explore both built-in objects and the expressions you can code to use them.

Why Use Built-in Types?

If you’ve used lower-level languages such as C or C++, you know that much of your work centers on implementing objects—also known as data structures—to represent the components in your application’s domain. You need to lay out memory structures, manage memory allocation, implement search and access routines, and so on. These chores are about as tedious (and error-prone) as they sound, and they usually distract from your program’s real goals. In typical Python programs, most of this grunt work goes away. Because Python provides powerful object types as an intrinsic part of the language, there’s usually no need to code object implementations before you start solving problems. In fact, unless you have a need for special processing that built-in types don’t provide, you’re almost always better off using a built-in object instead of implementing your own.

Python’s Core Data Types

1. Numbers
2. Strings
3. Lists
4. Dictionaries
5. Tuples
6. Sets
7. Files
8. Other core types (Booleans)
9. Program Unit types (Functions, modules, classes)

Numbers

Python supports different numerical types −

int (signed integers) − They are often called just integers or ints. They are positive or negative whole numbers with no decimal point. Integers in Python 3 are of unlimited size. Python 2 has two integer types - int and long. There is no 'long integer' in Python 3 anymore.

float (floating point real values) − Also called floats, they represent real numbers and are written with a decimal point dividing the integer and the fractional parts. Floats may also be in scientific notation, with E or e indicating the power of 10 (2.5e2 = 2.5 x 102 = 250).

complex (complex numbers) − are of the form a + bJ, where a and b are floats and J (or j) represents the square root of -1 (which is an imaginary number). The real part of the number is a, and the imaginary part is b. Complex numbers are not used much in Python programming.

Numbers in Python support the normal mathematical operations. For instance, the plus sign (+) performs addition, a star (*) is used for multiplication, and two stars (**) are used for exponentiation:

In [1]:

```
123 + 453 #Integer addidtion
```

Out[1]:

```
576
```

In [2]:

```
1.5 * 4 #Floating-point multiplication
```

Out[2]:

```
6.0
```

In [3]:

```
2 ** 100 # 2 to the power 100
```

Out[3]:

```
1267650600228229401496703205376
```

Python 3.X’s integer type automatically provides extra precision for large numbers like this when needed.

In [4]:

```
number = 0xA0F #Hexa-decimal
```

In [5]:

```
number
```

Out[5]:

```
2575
```

In [6]:

```
number = 0o37 #Octal
number
```

Out[6]:

```
31
```

In [7]:

```
5 / 10 #floating point division
```

Out[7]:

```
0.5
```

In [8]:

```
20 / 5
```

Out[8]:

```
4.0
```

In [9]:

```
24 // 5 #  # floor division discards the fractional part
```

Out[9]:

```
4
```

In [10]:

```
24 / 5
```

Out[10]:

```
4.8
```

In [11]:

```
24 % 5 # the % operator returns the remainder of the division
```

Out[11]:

```
4
```

In [12]:

```
5 * 3 + 2  # result * divisor + remainder
```

Out[12]:

```
17
```

In [13]:

```
width = 20
height = 5 * 9
width * height
```

Out[13]:

```
900
```

In [14]:

```
4 * 3.75 - 1
```

Out[14]:

```
14.0
```

In [15]:

```
tax = 12.5 / 100
price = 100.50
price * tax
```

Out[15]:

```
12.5625
```

In interactive mode, the last printed expression is assigned to the variable _. This means that when you are using Python as a desk calculator, it is somewhat easier to continue calculations, for example:

In [16]:

```
price + _
```

Out[16]:

```
113.0625
```

In [17]:

```
round(_, 2)
```

Out[17]:

```
113.06
```

Built-In Math Functions

[https://docs.python.org/3/library/math.html (https://docs.python.org/3/library/math.html)]

In [18]:

```
import math
```

In [19]:

```
math.pi
```

Out[19]:

```
3.141592653589793
```

In [20]:

```
math.sqrt(4)
```

Out[20]:

```
2.0
```

In [21]:

```
math.sqrt(85)
```

Out[21]:

```
9.219544457292887
```

In [22]:

```
math.ceil(2.4) #Return the ceiling of x, the smallest integer greater than or equal to
 x
```

Out[22]:

```
3
```

In [23]:

```
math.ceil(2.6)
```

Out[23]:

```
3
```

In [24]:

```
math.fabs(-4) #Return the absolute value of x.
```

Out[24]:

```
4.0
```

In [25]:

```
math.floor(2.8) #Return the floor of x, the largest integer less than or equal to x
```

Out[25]:

```
2
```

In [26]:

```
math.copysign(2, -5) #Return a float with the magnitude (absolute value) of x but the s
ign of y
```

Out[26]:

```
-2.0
```

In [27]:

```
10 % 2.4
```

Out[27]:

```
0.40000000000000036
```

In [28]:

```
math.fmod(10, 2.4) #fmod() is generally preferred when working with floats, while Pytho
n’s x % y is preferred when working with integers.
```

Out[28]:

```
0.40000000000000036
```

In [29]:

```
sum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1])
```

Out[29]:

```
0.9999999999999999
```

In [30]:

```
math.fsum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1])
# math.fsum(iterable)
#Return an accurate floating point sum of values in the iterable. Avoids loss of precis
ion by tracking multiple intermediate partial sums:
```

Out[30]:

```
1.0
```

In [31]:

```
math.gcd(10, 5)
```

Out[31]:

```
5
```

In [32]:

```
math.gcd(64, 2)
```

Out[32]:

```
2
```

Power and logarithmic functions

In [33]:

```
math.pow(2, 8) #Return x raised to the power y
```

Out[33]:

```
256.0
```

In [34]:

```
math.log(2)
```

Out[34]:

```
0.6931471805599453
```

In [35]:

```
math.log(2, 10)
```

Out[35]:

```
0.30102999566398114
```

In [36]:

```
math.exp(1e-5)      #gives result accurate to 11 places
```

Out[36]:

```
1.00001000005
```

In [37]:

```
math.expm1(1e-5)    # result accurate to full precision
```

Out[37]:

```
1.0000050000166667e-05
```

Trigonometric functions

In [38]:

```
math.cos(0) #Return the cosine of x radians.
```

Out[38]:

```
1.0
```

In [39]:

```
math.degrees(90) #Convert angle x from radians to degrees.
```

Out[39]:

```
5156.620156177409
```

In [40]:

```
math.cos(45)
```

Out[40]:

```
0.5253219888177297
```

In [41]:

```
math.sin(90)
```

Out[41]:

```
0.8939966636005579
```

In [42]:

```
math.sin(0)
```

Out[42]:

```
0.0
```

In [43]:

```
math.tanh(90)
```

Out[43]:

```
1.0
```

Strings

Strings are used to record both textual information (your name, for instance) as well as arbitrary collections of bytes (such as an image file's contents). They are our first example of what in Python we call a sequence—a positionally ordered collection of other objects. Sequences maintain a left-to-right order among the items they contain: their items are stored and fetched by their relative positions. Strictly speaking, strings are sequences of one-character strings; other, more general sequence types include lists and tuples, covered later

In [44]:

```
x = 'Hello'
```

In [45]:

```
x
```

Out[45]:

```
'Hello'
```

In [46]:

```
type(x)
```

Out[46]:

```
str
```

In [47]:

```
x = "Hello"
type(x)
```

Out[47]:

```
str
```

In [48]:

```
len(x) #length of a string
```

Out[48]:

```
5
```

In [49]:

```
x[0] #the first item in x, indexing by zero-based position
```

Out[49]:

```
'H'
```

In [50]:

```
x[1]
```

Out[50]:

```
'e'
```

In Python, indexes are coded as offsets from the front, and so start from 0: the first item is at index 0, the second is at index 1, and so on.

Python variables never need to be declared ahead of time. A variable is created when you assign it a value, may be assigned any type of object, and is replaced with its value when it shows up in an expression

In Python, we can also index backward, from the end—positive indexes count from the left, and negative indexes count back from the right:

In [51]:

```
x[-1]
```

Out[51]:

```
'o'
```

In [52]:

```
x[-4]
```

Out[52]:

```
'e'
```

Formally, a negative index is simply added to the string’s length, so the following two operations are equivalent (though the first is easier to code and less easy to get wrong):

In [53]:

```
x[-1]
```

Out[53]:

```
'o'
```

In [54]:

```
x[len(x) - 1]
```

Out[54]:

```
'o'
```

In [55]:

```
x[len(x) - 4]
```

Out[55]:

```
'e'
```

Notice that we can use an arbitrary expression in the square brackets, not just a hardcoded number literal—anywhere that Python expects a value, we can use a literal, a variable, or any expression we wish. Python’s syntax is completely general this way. In addition to simple positional indexing, sequences also support a more general form of indexing known as slicing, which is a way to extract an entire section (slice) in a single step. For example:

In [56]:

```
x
```

Out[56]:

```
'Hello'
```

In [57]:

```
x[1:3]  # Slice of x from offsets 1 through 2 (not 3)
```

Out[57]:

```
'el'
```

Their general form, X[I:J], means “give me everything in X from offset I up to but not including offset J.”

In [58]:

```
x[0:3]
```

Out[58]:

```
'Hel'
```

In [59]:

```
x[:]  # All of x as a top-level copy (0:len(S))
```

Out[59]:

```
'Hello'
```

In [60]:

```
x[:-1]
```

Out[60]:

```
'Hell'
```

In [61]:

```
x[:3]
```

Out[61]:

```
'Hel'
```

In [62]:

```
'doesn\'t' # use \' to escape the single quote...
```

Out[62]:

```
"doesn't"
```

In [63]:

```
"doesn't"
```

Out[63]:

```
"doesn't"
```

In [64]:

```
print('"Isn\'t," she said.')
```

```
"Isn't," she said.
```

In [65]:

```
s = 'First line.\nSecond line.'  # \n means newline
```

In [66]:

```
s
```

Out[66]:

```
'First line.\nSecond line.'
```

The print() function produces a more readable output, by omitting the enclosing quotes and by printing escaped and special characters:

In [67]:

```
print(s)
```

```
First line.
Second line.
```

If you don’t want characters prefaced by \ to be interpreted as special characters, you can use raw strings by adding an r before the first quote:

In [68]:

```
print('C:\some\name') # # here \n means newline!
```

```
C:\some
ame
```

In [69]:

```
print(r'C:\some\name') # note the r before the quote
```

```
C:\some\name
```

String literals can span multiple lines. One way is using triple-quotes: """...""" or '''...'''. End of lines are automatically included in the string, but it’s possible to prevent this by adding a \ at the end of the line. The following example:

In [70]:

```
print('''\
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
''')
```

```
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
```

Strings can be concatenated (glued together) with the + operator, and repeated with *:

In [71]:

```
3 * 'un' + 'ium'
```

Out[71]:

```
'unununium'
```

Two or more string literals (i.e. the ones enclosed between quotes) next to each other are automatically concatenated.

In [72]:

```
'Py' 'thon'
```

Out[72]:

```
'Python'
```

This feature is particularly useful when you want to break long strings:

In [73]:

```
text = ('Put several strings within parentheses '
        'to have them joined together.')
```

In [74]:

```
text
```

Out[74]:

```
'Put several strings within parentheses to have them joined together.'
```

This only works with two literals though, not with variables or expressions: prefix = 'Py' prefix 'thon' # can't concatenate a variable and a string literal

SyntaxError: invalid syntax ('un' * 3) 'ium'

If you want to concatenate variables or a variable and a literal, use +:

In [75]:

```
prefix = 'py'
prefix + 'thon'
```

Out[75]:

```
'python'
```

In [76]:

```
word = 'Python'
word[2] + word[4]
```

Out[76]:

```
'to'
```

Built-In String Functions

In [77]:

```
s = 'Sammy hark'
```

Making Strings Upper and Lower Case

In [78]:

```
s.upper()  #str.upper() and str.lower()
```

Out[78]:

```
'SAMMY HARK'
```

In [79]:

```
s.lower()
```

Out[79]:

```
'sammy hark'
```

In [80]:

```
s.capitalize()
```

Out[80]:

```
'Sammy hark'
```

In [81]:

```
s.swapcase() #This method swaps the case of every character i.e. every uppercase is con
verted to lowercase and vice versa.
```

Out[81]:

```
'sAMMY HARK'
```

In [82]:

```
s.title()
```

Out[82]:

```
'Sammy Hark'
```

In [83]:

```
print(s.index("S"))
```

```
0
```

In [84]:

```
print(s.count("m"))
```

```
2
```

In [85]:

```
print(s[0:5:2])
```

```
Smy
```

In [86]:

```
s[::-1]
```

Out[86]:

```
'krah ymmaS'
```

In [87]:

```
s.startswith('X')
```

Out[87]:

```
False
```

In [88]:

```
s.startswith('Sam')
```

Out[88]:

```
True
```

In [89]:

```
s.startswith('sam')
```

Out[89]:

```
False
```

In [90]:

```
s.endswith('ark')
```

Out[90]:

```
True
```

join(), split(), and replace() Methods

The str.join(), str.split(), and str.replace() methods are a few additional ways to manipulate strings in Python.

The str.join() method will concatenate two strings, but in a way that passes one string through another.

In [91]:

```
balloon = "Sammy has a balloon."
```

In [92]:

```
balloon.join('Hello')
```

Out[92]:

```
'HSammy has a balloon.eSammy has a balloon.lSammy has a balloon.lSammy has
a balloon.o'
```

In [93]:

```
balloon.join(' Hello')
```

Out[93]:

```
' Sammy has a balloon.HSammy has a balloon.eSammy has a balloon.lSammy has
a balloon.lSammy has a balloon.o'
```

In [94]:

```
' '.join(balloon)
```

Out[94]:

```
'S a m m y   h a s   a   b a l l o o n .'
```

In [95]:

```
' '.join(reversed(balloon))
```

Out[95]:

```
'. n o o l l a b   a   s a h   y m m a S'
```

In [96]:

```
print(','.join(["sharks", "crustaceans", "plankton"]))
```

```
sharks,crustaceans,plankton
```

In [97]:

```
print(', '.join(["sharks", "crustaceans", "plankton"]))
```

```
sharks, crustaceans, plankton
```

Just as we can join strings together, we can also split strings up. To do this, we will use the str.split() method:

In [98]:

```
balloon.split()
```

Out[98]:

```
['Sammy', 'has', 'a', 'balloon.']
```

In [99]:

```
balloon.split('a')
```

Out[99]:

```
['S', 'mmy h', 's ', ' b', 'lloon.']
```

In [100]:

```
s = '123'
i = iter(s)
```

In [101]:

```
i.__next__()
```

Out[101]:

```
'1'
```

In [102]:

```
i.__next__()
```

Out[102]:

```
'2'
```

In [103]:

```
i.__next__()
```

Out[103]:

```
'3'
```

In [104]:

```
list(s)
```

Out[104]:

```
['1', '2', '3']
```

Now the letter a has been removed and the strings have been separated where each instance of the letter a had been, with whitespace retained.

The **str.replace()** method can take an original string and return an updated string with some replacement.

Let’s say that the balloon that Sammy had is lost. Since Sammy no longer has this balloon, we will change the substring "has" from the original string balloon to "had" in a new string:

In [105]:

```
print(balloon.replace("has","had"))
```

```
Sammy had a balloon.
```

We search for a specific character or characters in a string with the .find() method.

In [106]:

```
s = "On the other hand, you have different fingers."
s.find("hand")
```

Out[106]:

```
13
```

In [107]:

```
s.find("o")
```

Out[107]:

```
7
```

In [108]:

```
s.find('o', 8)
```

Out[108]:

```
20
```

In [109]:

```
s.find("e", 20, -5)
```

Out[109]:

```
26
```

In [110]:

```
s.find('e')
```

Out[110]:

```
5
```

Python has a function called dir that lists the methods available for an object. The type function shows the type of an object and the dir function shows the available methods.

In [111]:

```
s = 'Hello'
```

In [112]:

```
type(s)
```

Out[112]:

```
str
```

In [113]:

```
dir(s)
```

Out[113]:

```
['__add__',
 '__class__',
 '__contains__',
 '__delattr__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__getnewargs__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
 '__mod__',
 '__mul__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__rmod__',
 '__rmul__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 'capitalize',
 'casefold',
 'center',
 'count',
 'encode',
 'endswith',
 'expandtabs',
 'find',
 'format',
 'format_map',
 'index',
 'isalnum',
 'isalpha',
 'isdecimal',
 'isdigit',
 'isidentifier',
 'islower',
 'isnumeric',
 'isprintable',
 'isspace',
 'istitle',
 'isupper',
 'join',
 'ljust',
 'lower',
 'lstrip',
 'maketrans'
 'maketrans',
 'partition',
 'replace',
 'rfind',
 'rindex',
 'rjust',
 'rpartition',
 'rsplit',
 'rstrip',
 'split',
 'splitlines',
 'startswith',
 'strip',
 'swapcase',
 'title',
 'translate',
 'upper',
 'zfill']
```

In [114]:

```
word = "    xyz    "
```

In [115]:

```
word.strip()
```

Out[115]:

```
'xyz'
```

In [116]:

```
word.lstrip()
```

Out[116]:

```
'xyz    '
```

In [117]:

```
word.rstrip()
```

Out[117]:

```
'    xyz'
```

In [118]:

```
'x' in word
```

Out[118]:

```
True
```

Strings are immutable

In [119]:

```
greeting = 'Hello, world!'
#greeting[0] = 'J'
```

In [120]:

```
new_greeting = 'J' + greeting[1:]
```

In [121]:

```
new_greeting
```

Out[121]:

```
'Jello, world!'
```

Lists

A list is a sequence Like a string, a list is a sequence of values. In a string, the values are characters; in a list, they can be any type. The values in a list are called elements or sometimes items. Thereareseveralwaystocreateanewlist;thesimplestistoenclosetheelementsinsquare brackets ([and]):

[10, 20, 30, 40]
['crunchy frog', 'ram bladder', 'lark vomit']

They are also mutable—unlike strings, lists can be modified in place by assignment to offsets as well as a variety of list method calls.

In [122]:

```
cheeses = ['Cheddar', 'Edam', 'Gouda']
```

In [123]:

```
cheeses
```

Out[123]:

```
['Cheddar', 'Edam', 'Gouda']
```

In [124]:

```
type(cheeses)
```

Out[124]:

```
list
```

In [125]:

```
 numbers = [17, 123]
```

In [126]:

```
numbers
```

Out[126]:

```
[17, 123]
```

In [127]:

```
 L = [123, 'spam', 1.23]
```

In [128]:

```
L[0]    #indexing by position
```

Out[128]:

```
123
```

In [129]:

```
L[:-1]  #list slicing
```

Out[129]:

```
[123, 'spam']
```

In [130]:

```
 L + [4, 5, 6]   #concatenating list
```

Out[130]:

```
[123, 'spam', 1.23, 4, 5, 6]
```

In [131]:

```
L * 2  # repeating a list. Here we are not changing the original list
```

Out[131]:

```
[123, 'spam', 1.23, 123, 'spam', 1.23]
```

In [132]:

```
L
```

Out[132]:

```
[123, 'spam', 1.23]
```

In [133]:

```
M = L * 2 #assigning the new list to a variable
```

In [134]:

```
M
```

Out[134]:

```
[123, 'spam', 1.23, 123, 'spam', 1.23]
```

Lists are mutable

In [135]:

```
M
```

Out[135]:

```
[123, 'spam', 1.23, 123, 'spam', 1.23]
```

In [136]:

```
M[4] = 'Hello!'
```

In [137]:

```
M
```

Out[137]:

```
[123, 'spam', 1.23, 123, 'Hello!', 1.23]
```

List Methods

list.append()
The method list.append(x) will add an item (x) to the end of a list. We'll start with a list of our fish that are dispersed throughout the aquarium.

In [138]:

```
fish = ['barracuda','cod','devil ray','eel']
```

In [139]:

```
fish.index('cod')
```

Out[139]:

```
1
```

In [140]:

```
fish.append('flounder')
```

In [141]:

```
print(fish)
```

```
['barracuda', 'cod', 'devil ray', 'eel', 'flounder']
```

In [142]:

```
fish.append('apple')
```

In [143]:

```
fish
```

Out[143]:

```
['barracuda', 'cod', 'devil ray', 'eel', 'flounder', 'apple']
```

list.insert()
The list.insert(i,x) method takes two arguments, with i being the index position you would like to add an item to, and x being the item itself.

In [144]:

```
fish.insert(0, 'anchony')
```

In [145]:

```
fish
```

Out[145]:

```
['anchony', 'barracuda', 'cod', 'devil ray', 'eel', 'flounder', 'apple']
```

list.extend()
If we want to combine more than one list, we can use the list.extend(L) method, which takes in a second list as its argument. Our aquarium is welcoming four new fish from another aquarium that is closing. We have these fish together in the list more_fish:

In [146]:

```
more_fish = ['goby','herring','ide','kissing gourami']
```

In [147]:

```
fish.extend(more_fish)
```

In [148]:

```
fish
```

Out[148]:

```
['anchony',
 'barracuda',
 'cod',
 'devil ray',
 'eel',
 'flounder',
 'apple',
 'goby',
 'herring',
 'ide',
 'kissing gourami']
```

In [149]:

```
len(fish)
```

Out[149]:

```
11
```

list.remove()
When we need to remove an item from a list, we'll use the list.remove(x) method which removes the first item in a list whose value is equivalent to x.

In [150]:

```
fish.remove('devil ray')
```

In [151]:

```
fish
```

Out[151]:

```
['anchony',
 'barracuda',
 'cod',
 'eel',
 'flounder',
 'apple',
 'goby',
 'herring',
 'ide',
 'kissing gourami']
```

In [152]:

```
'devil ray' in fish
```

Out[152]:

```
False
```

list.pop()
We can use the list.pop([i]) method to return the item at the given index position from the list and then remove that item. The square brackets around the i for index tell us that this parameter is optional, so if we don’t specify an index (as in fish.pop()), the last item will be returned and removed.

In [153]:

```
fish.pop(4)
```

Out[153]:

```
'flounder'
```

In [154]:

```
fish
```

Out[154]:

```
['anchony',
 'barracuda',
 'cod',
 'eel',
 'apple',
 'goby',
 'herring',
 'ide',
 'kissing gourami']
```

list.copy()
When we are working with a list and may want to manipulate it in multiple ways while still having the original list available to us unchanged, we can use list.copy() to make a copy of the list. We'll pass the value returned from fish.copy() to the variable fish_2

```
In [155]:
```

```
fish_2 = fish.copy()
```

```
In [156]:
```

```
fish_2
```

```
Out[156]:

['anchony',
 'barracuda',
 'cod',
 'eel',
 'apple',
 'goby',
 'herring',
 'ide',
 'kissing gourami']
```

```
In [157]:
```

```
fish.reverse()
```

```
In [158]:
```

```
fish
```

```
Out[158]:

['kissing gourami',
 'ide',
 'herring',
 'goby',
 'apple',
 'eel',
 'cod',
 'barracuda',
 'anchony']
```

list.count()
The list.count(x) method will return the number of times the value x occurs within a specified list.

```
In [159]:
```

```
fish.count('goby')
```

```
Out[159]:

1
```

list.sort()
We can use the list.sort() method to sort the items in a list.

In [160]:

```
fish.sort()
```

In [161]:

```
fish
```

Out[161]:

```
['anchony',
 'apple',
 'barracuda',
 'cod',
 'eel',
 'goby',
 'herring',
 'ide',
 'kissing gourami']
```

list.clear()
When we’re done with a list, we can remove all values contained in it by using the list.clear() method.

In [162]:

```
fish.clear()
```

In [163]:

```
fish
```

Out[163]:

```
[]
```

Nesting

In [164]:

```
M = [
    [1, 2, 3],
    [5.2, 6.3,  'python']
]
```

In [165]:

```
M[0]
```

Out[165]:

```
[1, 2, 3]
```

In [166]:

```
M[1]
```

Out[166]:

```
[5.2, 6.3, 'python']
```

In [167]:

```
M[0][1]
```

Out[167]:

```
2
```

In [168]:

```
M[0][1] = 'programming'
```

In [169]:

```
M
```

Out[169]:

```
[[1, 'programming', 3], [5.2, 6.3, 'python']]
```

Astringisasequenceofcharactersandalistisasequenceofvalues,butalistofcharacters isnotthesameasastring. Toconvertfromastringtoalistofcharacters,youcanuse list:

String and List

In [170]:

```
s = 'spam'
```

In [171]:

```
t = list(s)
```

In [172]:

```
t
```

Out[172]:

```
['s', 'p', 'a', 'm']
```

In [173]:

```
 s = 'pining for the fjords'
```

In [174]:

```
s.split()
```

Out[174]:

```
['pining', 'for', 'the', 'fjords']
```

In [175]:

```
a = [1, 2, 3]
b = [1, 2, 3]
```

To check whether two variables refer to the same object, you can use the is operator.

In [176]:

```
a is  b
```

Out[176]:

```
False
```

In the above case we would say that the two lists are equivalent, because they have the same elements,butnotidentical,becausetheyarenotthesameobject. Iftwoobjectsareidentical, they are also equivalent, but if they are equivalent, they are not necessarily identical.

In [177]:

```
c = 'banana'
d = 'banana'
```

In [178]:

```
c is d
```

Out[178]:

```
True
```

In this example, Python only created one string object, and both a and b refer to it.

In [179]:

```
id(c)   #id(object) : As we can see the function accepts a single parameter and is used
 to return the identity of an object. This identity has to be unique and constant for t
his object during the lifetime. Two objects with non-overlapping lifetimes may have the
 same id() value. If we relate this to C, then they are actually the memory address, he
re in Python it is the unique id. This function is generally used internally in Python.
```

Out[179]:

```
2566182468944
```

In [180]:

```
id(d)
```

Out[180]:

```
2566182468944
```

In [181]:

```
id(a), id(b)
```

Out[181]:

```
(2566182447560, 2566182475208)
```

Aliasing

If a refers to an object and you assign b = a, then both variables refer to the same object:

In [182]:

```
a = [1, 2, 3]
```

In [183]:

```
b = a
```

In [184]:

```
a is b
```

Out[184]:

```
True
```

In [185]:

```
b is a
```

Out[185]:

```
True
```

The association of a variable with an object is called a reference. In this example, there are two references to the same object. An object with more than one reference has more than one name, so we say that the object is aliased.

If the aliased object is mutable, changes made with one alias affect the other:

In [186]:

```
b[0] = 17
```

In [187]:

```
a
```

Out[187]:

```
[17, 2, 3]
```

Dictionaries

A dictionary is like a list, but more general. In a list, the indices have to be integers; in a dictionary they can be (almost) any type. You can think of adictionary as a mapping between a set of indices(which are called keys) and a set of values. Each key maps to a value. The association of a key and a value is called a key-value pair or sometimes an item.

As an example, we’ll build a dictionary that maps from English to Spanish words, so the keys and the values are all strings. The function dict creates a new dictionary with no items. Because dict is the name of a built-in function, you should avoid using it as a variable name.

In [188]:

```
eng2sp = dict()
```

In [189]:

```
eng2sp
```

Out[189]:

```
{}
```

The squiggly-brackets, {}, represent an empty dictionary. To add items to the dictionary, you can use square brackets:

In [190]:

```
eng2sp['one'] = 'uno'
```

This line creates an item that maps from the key 'one' to the value 'uno'. If we print the dictionary again, we see a key-value pair with a colon between the key and value:

In [191]:

```
eng2sp
```

Out[191]:

```
{'one': 'uno'}
```

In [192]:

```
eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

In [193]:

```
eng2sp
```

Out[193]:

```
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

The order of the key-value pairs is not the same. In fact, if you type the same example on your computer, you might get a different result. In general, the order of items in a dictionary is unpredictable.

In [194]:

```
eng2sp['two']
```

Out[194]:

```
'dos'
```

In [195]:

```
len(eng2sp)
```

Out[195]:

```
3
```

In [196]:

```
'one' in eng2sp
```

Out[196]:

```
True
```

In [197]:

```
'uno' in eng2sp
```

Out[197]:

```
False
```

The in operator works on dictionaries; it tells you whether something appears as a key in the dictionary.

In [198]:

```
'uno' in eng2sp.values()
```

Out[198]:

```
True
```

In [199]:

```
eng2sp.values()
```

Out[199]:

```
dict_values(['uno', 'dos', 'tres'])
```

In [200]:

```
eng2sp.keys()
```

Out[200]:

```
dict_keys(['one', 'two', 'three'])
```

The ino perator uses different algorithms for lists and dictionaries. For lists,it uses a search algorithm. As the list gets longer, the search time gets longer in direct proportion. For dictionaries, Python uses an algorithm called a hashtable that has a remarkable property: the in operator takes about the same amount of time no matter how many items there are in a dictionary. http://en.wikipedia.org/wiki/Hash_table. (http://en.wikipedia.org/wiki/Hash_table.)

In [201]:

```
eng2sp.items()
```

Out[201]:

```
dict_items([('one', 'uno'), ('two', 'dos'), ('three', 'tres')])
```

In [202]:

```
eng2sp.get('one')
```

Out[202]:

```
'uno'
```

Updating Values in dictionary

In [203]:

```
sammy = {'username': 'sammy-shark', 'online': True, 'followers': 987}
```

In [204]:

```
sammy['username'] = 'python'
```

In [205]:

```
sammy
```

Out[205]:

```
{'followers': 987, 'online': True, 'username': 'python'}
```

In [206]:

```
sammy.update({'online': False})
```

In [207]:

```
sammy
```

Out[207]:

```
{'followers': 987, 'online': False, 'username': 'python'}
```

In [208]:

```
sammy.update({'Gender': 'Male'})  # adding new item
```

In [209]:

```
sammy
```

Out[209]:

```
{'Gender': 'Male', 'followers': 987, 'online': False, 'username': 'pytho
n'}
```

Deleting Dictionary Elements

In [210]:

```
del sammy['followers']
```

In [211]:

```
sammy
```

Out[211]:

```
{'Gender': 'Male', 'online': False, 'username': 'python'}
```

getting key from the given value:

In [212]:

```
mydict = {'george':16,'amber':19}
```

In [213]:

```
mydict.values()
```

Out[213]:

```
dict_values([16, 19])
```

In [214]:

```
mydict.keys()
```

Out[214]:

```
dict_keys(['george', 'amber'])
```

In [215]:

```
list(mydict.values())
```

Out[215]:

```
[16, 19]
```

In [216]:

```
list(mydict.values()).index(16)
```

Out[216]:

```
0
```

In [217]:

```
list(mydict.values())[0]
```

Out[217]:

```
16
```

In [218]:

```
list(mydict.keys()).index('george')
```

Out[218]:

```
0
```

In [219]:

```
list(mydict.keys())[0]
```

Out[219]:

```
'george'
```

In [220]:

```
[list(mydict.values()).index(16)]
```

Out[220]:

```
[0]
```

In [221]:

```
list(mydict.keys())[list(mydict.values()).index(16)]
```

Out[221]:

```
'george'
```

Deleting the key-value pairs based on the given values

In [222]:

```
del mydict[list(mydict.keys())[list(mydict.values()).index(16)]]
```

In [223]:

```
mydict
```

Out[223]:

```
{'amber': 19}
```

In [224]:

```
mydict.clear()
```

In [225]:

```
mydict
```

Out[225]:

```
{}
```

In [226]:

```
dir(mydict)
```

Out[226]:

```
['__class__',
 '__contains__',
 '__delattr__',
 '__delitem__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__setitem__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 'clear',
 'copy',
 'fromkeys',
 'get',
 'items',
 'keys',
 'pop',
 'popitem',
 'setdefault',
 'update',
 'values']
```

fromkeys()

The fromkeys() method creates a new dictionary from the given sequence of elements with a value provided by the user.

The syntax of fromkeys() method is:
dictionary.fromkeys(sequence[, value])

sequence - sequence of elements which is to be used as keys for the new dictionary
value (Optional) - value which is set to each each element of the dictionary

Example 1: Create a dictionary from a sequence of keys:

In [227]:

```
# vowels keys
keys = {'a', 'e', 'i', 'o', 'u' }

vowels = dict.fromkeys(keys)
print(vowels)
```

```
{'a': None, 'o': None, 'i': None, 'u': None, 'e': None}
```

Example 2: Create a dictionary from a sequence of keys with value

In [228]:

```
# vowels keys
keys = {'a', 'e', 'i', 'o', 'u' }
value = 'vowel'

vowels = dict.fromkeys(keys, value)
print(vowels)
```

```
{'a': 'vowel', 'o': 'vowel', 'i': 'vowel', 'u': 'vowel', 'e': 'vowel'}
```

In [229]:

```
keys = {'a', 'e', 'i', 'o', 'u' }
value = 'vowel'

vowels = dict.fromkeys(keys, value)
print(vowels)
```

```
{'a': 'vowel', 'o': 'vowel', 'i': 'vowel', 'u': 'vowel', 'e': 'vowel'}
```

Example 3: Create a dictionary from mutable object list

In [230]:

```
# vowels keys
keys = {'a', 'e', 'i', 'o', 'u' }
value = [1]

vowels = dict.fromkeys(keys, value)
print(vowels)

# updating the value
value.append(2)
print(vowels)
```

```
{'a': [1], 'o': [1], 'i': [1], 'u': [1], 'e': [1]}
{'a': [1, 2], 'o': [1, 2], 'i': [1, 2], 'u': [1, 2], 'e': [1, 2]}
```

If the provided value is a mutable object (whose value can be modified) like list, dictionary, etc., when the mutable object is modified, each element of the sequence also gets updated. This is because, each element is assigned a reference to the same object (points to the same object in the memory). To avoid this issue, we use dictionary comprehension.

In [231]:

```
# vowels keys
keys = {'a', 'e', 'i', 'o', 'u' }
value = [1]

vowels = { key : list(value) for key in keys }
# you can also use { key : value[:] for key in keys }
print(vowels)

# updating the value
value.append(2)
print(vowels)
```

```
{'a': [1], 'o': [1], 'i': [1], 'u': [1], 'e': [1]}
{'a': [1], 'o': [1], 'i': [1], 'u': [1], 'e': [1]}
```

In [232]:

```
value
```

Out[232]:

```
[1, 2]
```

In [233]:

```
vowels
```

Out[233]:

```
{'a': [1], 'e': [1], 'i': [1], 'o': [1], 'u': [1]}
```

In [234]:

```
vowels['a'].append('apples')
```

In [235]:

```
vowels
```

Out[235]:

```
{'a': [1, 'apples'], 'e': [1], 'i': [1], 'o': [1], 'u': [1]}
```

In [236]:

```
vowels['a'].remove(1)
```

In [237]:

```
vowels
```

Out[237]:

```
{'a': ['apples'], 'e': [1], 'i': [1], 'o': [1], 'u': [1]}
```

Remove, delete and pop

In [238]:

```
a=[1,2,3]
a.remove(2)
```

In [239]:

```
a
```

Out[239]:

```
[1, 3]
```

In [240]:

```
a=[1,2,3]
del a[1]
```

In [241]:

```
a
```

Out[241]:

```
[1, 3]
```

In [242]:

```
a= [1,2,3]
a.pop(1)
```

Out[242]:

```
2
```

In [243]:

```
a
```

Out[243]:

```
[1, 3]
```

remove removes the first matching value, not a specific index.

del removes a specific index.

and pop returns the removed element.

In [244]:

```
a={1:"a", 3:"b", 5:"c"}
```

In [245]:

```
a
```

Out[245]:

```
{1: 'a', 3: 'b', 5: 'c'}
```

In [246]:

```
a.popitem()
```

Out[246]:

```
(5, 'c')
```

In [247]:

```
a
```

Out[247]:

```
{1: 'a', 3: 'b'}
```

In [248]:

```
a.pop(1)
```

Out[248]:

```
'a'
```

In [249]:

```
a
```

Out[249]:

```
{3: 'b'}
```

Lists from Dictionaries

In [250]:

```
w = {"house":"Haus", "cat":"", "red":"rot"}
```

In [251]:

```
items_view = w.items()
```

In [252]:

```
items = list(items_view)
```

In [253]:

```
items
```

Out[253]:

```
[('house', 'Haus'), ('cat', ''), ('red', 'rot')]
```

Turn Lists into Dictionaries

In [254]:

```
dishes = ["pizza", "sauerkraut", "paella", "hamburger"]
countries = ["Italy", "Germany", "Spain", "USA"]
```