**2.1 Strings**

A string is a *sequence* of characters. You can access the characters one at a time with the bracket operator:

>>> fruit = 'banana'

>>> letter = fruit[1]

The second statement extracts the character at index position 1 from the fruit variable and assigns it to the letter variable.

The expression in brackets is called an *index*. The index indicates which character in the sequence you want (hence the name).

But you might not get what you expect:

>>>print(letter)

a

For most people, the first letter of “banana” is b, not a. But in Python, the index is an offset from the beginning of the string, and the offset of the first letter is zero.

>>> letter = fruit[0]

>>>print(letter)

b

So b is the 0th letter (“zero-eth”) of “banana”, a is the 1th letter (“one-eth”), and n is the 2th (“two-eth”) letter.

You can use any expression, including variables and operators, as an index, but the value of the index has to be an integer. Otherwise you get:

>>> letter = fruit[1.5]

TypeError: string indices must be integers

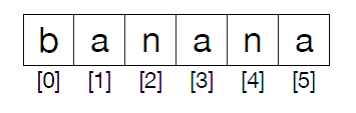


Figure 2.1: String Indexes

**2.2 String Operations**

**Getting the length of a string using len**

len is a built-in function that returns the number of characters in a string:

>>> fruit = 'banana'

>>>len(fruit)

6

To get the last letter of a string, you might be tempted to try something like this:

>>> length = len(fruit)

>>> last = fruit[length]

IndexError: string index out of range

The reason for the IndexError is that there is no letter in ’banana’ with the index 6. Since we started counting at zero, the six letters are numbered 0 to 5. To get the last character, you have to subtract 1 from length:

>>> last = fruit[length-1]

>>>print(last)

a

**The in operator**

The word **in** is a boolean operator that takes two strings and returns True if the first appears as a substring in the second:

>>>'a' in 'banana'

True

>>>'seed' in 'banana'

False

**String comparison**

The comparison operators work on strings. To see if two strings are equal:

**if** word == 'banana':

print('All right, bananas.')

Other comparison operations are useful for putting words in alphabetical order:

**if** word <'banana':

print('Your word,' + word + ', comes before banana.')

**elif** word >'banana':

print('Your word,' + word + ', comes after banana.')

**else**:

print('All right, bananas.')

Python does not handle uppercase and lowercase letters the same way that people do. All the uppercase letters come before all the lowercase letters, so:

Your word, Pineapple, comes before banana.

A common way to address this problem is to convert strings to a standard format, such as all lowercase, before performing the comparison. Keep that in mind in case you have to defend yourself against a man armed with a Pineapple.

**2.3 Accessing Characters in String by Index Number**

A lot of computations involve processing a string one character at a time. Often they start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a *traversal*. One way to write a traversal is with a while loop:

index = 0

**while** index <len(fruit):

letter = fruit[index]

print(letter)

index = index + 1

This loop traverses the string and displays each letter on a line by itself. The loop condition is index \< len(fruit), so when index is equal to the length of the string, the condition is false, and the body of the loop is not executed. The last character accessed is the one with the index len(fruit)-1, which is the last character in the string.

Another way to write a traversal is with a for loop:

**for** char in fruit:

print(char)

Each time through the loop, the next character in the string is assigned to the variable char. The loop continues until no characters are left.

**2.4 String Slicing and Joining**

A segment of a string is called a *slice*. Selecting a slice is similar to selecting a character:

>>> s = 'Monty Python'

>>>print(s[0:5])

Monty

>>>print(s[6:12])

Python

The operator returns the part of the string from the “n-eth” character to the “m-eth” character, including the first but excluding the last.

If you omit the first index (before the colon), the slice starts at the beginning of the string. If you omit the second index, the slice goes to the end of the string:

>>> fruit = 'banana'

>>> fruit[:3]

*'ban'*

>>> fruit[3:]

*'ana'*

If the first index is greater than or equal to the second the result is an *empty string*, represented by two quotation marks:

>>> fruit = 'banana'

>>> fruit[3:3]

*''*

An empty string contains no characters and has length 0, but other than that, it is the same as any other string.

**Slicing with a Step**

A Step can be used to slice a string as well

>>> s = 'Monty Python'

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

Mny

>>>print(s[6:12:2])

Pto

**Reverse Slicing**

Alternatively, you can use negative indices, which count backward from the end of the string.

A Negative Step and reversing the start and end indices can be used to reverse slice a String

>>> s = 'Monty Python'

>>>print(s[5:0:-1])

ytnoM

**Partial Slicing**

Start and End indices are optional here, If the starting Index is not specified the substring will brgin from the start, If ending Index is not specified, the substring will go all the way till the end.

**Joining Strings**

To join two strings together and also the same string to itself multiple times + and \* operator can be used respectively.

>>> s = 'Monty'

>>> t = 'Python'

>>>u=s+t

>>>print("Concatenated Strings : ",u)

>>>Concatenated Strings : MontyPython

The \* operator can be used to append the same string to itself

>>> s = 'Hello'

>>> s = s\*3

>>>print("Concatenated Same Strings : ",s)

>>> Concatenated Same Strings : HelloHelloHello

**Join() Method**

The Join() String method returns a string by joining all the elements of a list or a string, separated by a string operator.

>>> Text = ['Python','is','Fun']

>>> print(' '.join(Text))

>>> 'Python is Fun

**Strings are immutable**

It is tempting to use the operator on the left side of an assignment, with the intention of changing a character in a string. For example:

>>> greeting = 'Hello, world!'

>>> greeting[0] = 'J'

TypeError: 'str' object does not support item assignment

The “object” in this case is the string and the “item” is the character you tried to assign. For now, an *object* is the same thing as a value, but we will refine that definition later. An *item* is one of the values in a sequence.

The reason for the error is that strings are *immutable*, which means you can’t change an existing string. The best you can do is create a new string that is a variation on the original:

>>> greeting = 'Hello, world!'

>>> new\_greeting = 'J' + greeting[1:]

>>>print(new\_greeting)

Jello, world!

This example concatenates a new first letter onto a slice of greeting. It has no effect on the original string.

**Looping and counting**

The following program counts the number of times the letter a appears in a string:

word = 'banana'

count = 0

**for** letter in word:

**if** letter == 'a':

count = count + 1

print(count)

This program demonstrates another pattern of computation called a *counter*. The variable count is initialized to 0 and then incremented each time an a is found. When the loop exits, count contains the result: the total number of a’s.

**2.5 string methods**

Strings are an example of Python *objects*. An object contains both data (the actual string itself) and *methods*, which are effectively functions that are built into the object and are available to any *instance* of the object.

Python has a function called dir which lists the methods available for an object.

The type function shows the type of an object and the dir function shows the available methods.

>>> stuff = 'Hello world'

>>>type(stuff)

<**class** 'str'>

>>>dir(stuff)

['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', 'partition', 'replace', 'rfind',

*'rindex'* , 'rjust', 'rpartition', 'rsplit', 'rstrip',

*'split'* , 'splitlines', 'startswith', 'strip', 'swapcase',

*'title'* , 'translate', 'upper', 'zfill']

>>>help(str.capitalize)

Help on method\_descriptor:

capitalize(...)

S.capitalize() ->str

Return a capitalized version of S, i.e. make the first character

have upper case and the rest lower case.

>>>

While the dir function lists the methods, and you can use help to get some simple documentation on a method, a better source of documentation for string methods would be <https://docs.python.org/3.5/library/stdtypes.html#string-methods>.

Calling a *method* is similar to calling a function (it takes arguments and returns a value) but the syntax is different. We call a method by appending the method name to the variable name using the period as a delimiter.

For example, the method upper takes a string and returns a new string with all uppercase letters:

Instead of the function syntax upper(word), it uses the method syntax

word.upper().

>>> word = 'banana'

>>> new\_word = word.upper()

>>>print(new\_word)

BANANA

This form of dot notation specifies the name of the method, upper, and the name of the string to apply the method to, word. The empty parentheses indicate that this method takes no argument.

A method call is called an *invocation*; in this case, we would say that we are invoking upper on the word.

For example, there is a string method named find that searches for the position of one string within another:

>>> word = 'banana'

>>> index = word.find('a')

>>>print(index)

1

In this example, we invoke find on word and pass the letter we are looking for as a parameter.

The find method can find substrings as well as characters:

>>> word.find('na')

2

It can take as a second argument the index where it should start:

>>> word.find('na', 3)

4

One common task is to remove white space (spaces, tabs, or newlines) from the beginning and end of a string using the strip method:

>>> line = ' Here we go '

>>> line.strip()

*'Here we go'*

Some methods such as *startswith* return boolean values.

>>> line = 'Have a nice day'

>>> line.startswith('Have')

True

>>> line.startswith('h')

False

You will note that startswith requires case to match, so sometimes we take a line and map it all to lowercase before we do any checking using the lower method.

>>> line = 'Have a nice day'

>>> line.startswith('h')

False

>>> line.lower()

*'have a nice day'*

>>> line.lower().startswith('h')

True

In the last example, the method lower is called and then we use startswith to see if the resulting lowercase string starts with the letter “h”. As long as we are careful with the order, we can make multiple method calls in a single expression.

**2.6 Formatting strings**

It is the process of inserting a custom string or variable in predefined text.We put placeholders defined by a pair of curly braces in a text. We call the string dot format method. Then, we pass the desired value into the method. The method replaces the placeholders using the values in the order of appearance replace by value

**Methods for formatting**

We define a string and insert two placeholders. We pass two strings to the method, which will be passed to get the following output

>>>print("Machine learning provides {} the ability to learn {}".format("systems", "automatically"))

>>>Machine learning provides systems the ability to learn automatically

We can use variables for both the string and the values passed to the method

>>>my\_string = "{} rely on {} datasets"

>>>method = "Supervised algorithms"

>>>condition = "labeled"

>>>print(my\_string.format(method, condition))

>>>Supervised algorithms rely on labeled datasets

**Reordering Values**

In the below example, you add index numbers into the placeholders to reorder values. This affects the order in which the method replaces the placeholders.

>>>print("{} has a friend called {} and a sister called {}". format("Betty", "Linda", "Daisy"))

>>>Betty has a friend called Linda and a sister called Daisy

>>>print("{2} has a friend called {0} and a sister called {1}". format("Betty", "Linda", "Daisy"))

>>>Daisy has a friend called Betty and a sister called Linda

**Name Placeholders**

We can also introduce keyword arguments that are called by their keyword name.

>>>tool="Unsupervised algorithms"

>>>goal="patterns"

>>>print("{title} try to find {aim} in the dataset".format(title=tool, aim=goal))

>>>Unsupervised algorithms try to find patterns in the dataset

**Format Specifier**

The *format specifier*, % allows us to construct strings, replacing parts of the strings with the data stored in variables. When applied to integers, % is the modulus operator. But when the first operand is a string, % is the format operator.

The first operand is the *format string*, which contains one or more *format sequences* that specify how the second operand is formatted. The result is a string.

For example, the format sequence “%d” means that the second operand should be formatted as an integer (d stands for “decimal”):

>>> camels = 42

>>>'%d' % camels

*'42'*

The result is the string “42”, which is not to be confused with the integer value 42.

A format sequence can appear anywhere in the string, so you can embed a value in a sentence:

>>> camels = 42

>>>'I have spotted %d camels.' % camels

*'I have spotted 42 camels.'*

If there is more than one format sequence in the string, the second argument has to be a tuple1. Each format sequence is matched with an element of the tuple, in order.

The following example uses “%d” to format an integer, “%g” to format a floating point number (don’t ask why), and “%s” to format a string:

>>>'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')

*'In 3 years I have spotted 0.1 camels.'*

The number of elements in the tuple must match the number of format sequences in the string. The types of the elements also must match the format sequences:

>>>'%d %d %d' % (1, 2)

TypeError: not enough arguments **for** format string

>>>'%d' % 'dollars'

TypeError: %d format: a number is required, not str

In the first example, there aren’t enough elements; in the second, the element is the wrong type.

The format operator is powerful, but it can be difficult to use. You can read more about it at

<https://docs.python.org/3.5/library/stdtypes.html#printf-style-string-formatting>.

**2.7 Lists**

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 list are called *elements* or sometimes *items*.

There are several ways to create a new list; the simplest is to enclose the elements in square brackets ([ and ]):

[10, 20, 30, 40]

['crunchy frog', 'ram bladder', 'lark vomit']

The first example is a list of four integers. The second is a list of three strings. The elements of a list don’t have to be the same type. The following list contains a string, a float, an integer, and (lo!) another list:

['spam', 2.0, 5, [10, 20]]

A list within another list is *nested*. A list that contains no elements is called an empty list; you can create one with empty brackets, [].

As you might expect, you can assign list values to variables:

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

>>> numbers = [17, 123]

>>> empty = []

>>>print(cheeses, numbers, empty)

['Cheddar', 'Edam', 'Gouda'] [17, 123] []

**Lists are mutable**

The syntax for accessing the elements of a list is the same as for accessing the characters of a string: the bracket operator. The expression inside the brackets specifies the index. Remember that the indices start at 0:

>>>print(cheeses[0])

Cheddar

Unlike strings, lists are mutable because you can change the order of items in a list or reassign an item in a list. When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned.

>>> numbers = [17, 123]

>>> numbers[1] = 5

>>>print(numbers)

[17, 5]

The one-eth element of numbers, which used to be 123, is now 5.

You can think of a list as a relationship between indices and elements. This relationship is called a *mapping*; each index “maps to” one of the elements.

List indices work the same way as string indices:

• Any integer expression can be used as an index.

• If you try to read or write an element that does not exist, you get an

IndexError.

• If an index has a negative value, it counts backward from the end of the list.

The in operator also works on lists.

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

>>>'Edam' in cheeses

True

>>>'Brie' in cheeses

False

**Traversing a list**

The most common way to traverse the elements of a list is with a for loop. The syntax is the same as for strings:

**>>>for** cheese in cheeses:

>>>print(cheese)

This works well if you only need to read the elements of the list. But if you want to write or update the elements, you need the indices. A common way to do that is to combine the functions range and len:

**>>>for** i in range(len(numbers)):

**>>>**numbers[i] = numbers[i] \* 2

This loop traverses the list and updates each element. len returns the number of elements in the list. range returns a list of indices from 0 to *n* − 1, where *n* is the length of the list. Each time through the loop, i gets the index of the next element. The assignment statement in the body uses i to read the old value of the element and to assign the new value.

A for loop over an empty list never executes the body:

**>>>for** x in empty:

>>>print('This never happens.')

Although a list can contain another list, the nested list still counts as a single element. The length of this list is four:

['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]

**2.8 List operations**

The + operator concatenates lists:

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

>>> b = [4, 5, 6]

>>> c = a + b

>>>print(c)

>>>[1, 2, 3, 4, 5, 6]

Similarly, the \* operator repeats a list a given number of times:

>>> [0] \* 4

>>>[0, 0, 0, 0]

>>> [1, 2, 3] \* 3

>>>[1, 2, 3, 1, 2, 3, 1, 2, 3]

The first example repeats four times. The second example repeats the list three times.

**2.9 List slices**

The slice operator also works on lists:

>>> t = ['a', 'b', 'c', 'd', 'e', 'f']

>>> t[1:3]

>>>['b', 'c']

>>> t[:4]

>>>['a', 'b', 'c', 'd']

>>> t[3:]

>>>['d', 'e', 'f']

If you omit the first index, the slice starts at the beginning. If you omit the second, the slice goes to the end. So if you omit both, the slice is a copy of the whole list.

>>> t[:]

>>>['a', 'b', 'c', 'd', 'e', 'f']

Since lists are mutable, it is often useful to make a copy before performing operations that fold, spindle, or mutilate lists.

A slice operator on the left side of an assignment can update multiple elements:

>>> t = ['a', 'b', 'c', 'd', 'e', 'f']

>>> t[1:3] = ['x', 'y']

>>>print(t)

>>>['a', 'x', 'y', 'd', 'e', 'f']

**2.10 List methods**

Python provides methods that operate on lists. For example, append adds a new element to the end of a list:

>>> t = ['a', 'b', 'c']

>>> t.append('d')

>>>print(t)

['a', 'b', 'c', 'd']

extend takes a list as an argument and appends all of the elements:

>>> t1 = ['a', 'b', 'c']

>>> t2 = ['d', 'e']

>>> t1.extend(t2)

>>>print(t1)

['a', 'b', 'c', 'd', 'e']

This example leaves t2 unmodified.

sort arranges the elements of the list from low to high:

>>> t = ['d', 'c', 'e', 'b', 'a']

>>> t.sort()

>>>print(t)

['a', 'b', 'c', 'd', 'e']

Most list methods are void; they modify the list and return None. If you accidentally

write t = t.sort(), you will be disappointed with the result.

**Deleting elements**

There are several ways to delete elements from a list. If you know the index of the element you want, you can use pop:

>>> t = ['a', 'b', 'c']

>>> x = t.pop(1)

>>>print(t)

['a', 'c']

>>>print(x)

b

pop modifies the list and returns the element that was removed. If you don’t provide an index, it deletes and returns the last element.

If you don’t need the removed value, you can use the del operator:

>>> t = ['a', 'b', 'c']

>>>**del** t[1]

>>>print(t)

['a', 'c']

If you know the element you want to remove (but not the index), you can use remove:

>>> t = ['a', 'b', 'c']

>>> t.remove('b')

>>>print(t)

['a', 'c']

The return value from remove is None.

To remove more than one element, you can use del with a slice index:

>>> t = ['a', 'b', 'c', 'd', 'e', 'f']

>>>**del** t[1:5]

>>>print(t)

['a', 'f']

As usual, the slice selects all the elements up to, but not including, the second index.

**Lists and functions**

There are a number of built-in functions that can be used on lists that allow you to quickly look through a list without writing your own loops:

>>> nums = [3, 41, 12, 9, 74, 15]

>>>print(len(nums))

6

>>>print(max(nums))

74

>>>print(min(nums))

3

>>>print(sum(nums))

154

>>>print(sum(nums)/len(nums))

25

The sum() function only works when the list elements are numbers. The other functions (max(), len(), etc.) work with lists of strings and other types that can be comparable.

We could rewrite an earlier program that computed the average of a list of numbers entered by the user using a list.

First, the program to compute an average without a list:

total = 0

count = 0

**while** (True):

inp = input('Enter a number: ')

**if** inp == 'done': **break**

value = float(inp)

total = total + value

count = count + 1

average = total / count

print('Average:', average)

*# Code:* [*http://www.py4e.com/code3/avenum.py*](http://www.py4e.com/code3/avenum.py)

In this program, we have count and total variables to keep the number and running total of the user’s numbers as we repeatedly prompt the user for a number. We could simply remember each number as the user entered it and use built-in functions to compute the sum and count at the end.

numlist = list()

**while** (True):

inp = input('Enter a number: ')

**if** inp == 'done': **break**

value = float(inp)

numlist.append(value)

average = sum(numlist) / len(numlist)

print('Average:', average)

*# Code:* [*http://www.py4e.com/code3/avelist.py*](http://www.py4e.com/code3/avelist.py)

We make an empty list before the loop starts, and then each time we have a number, we append it to the list. At the end of the program, we simply compute the sum of the numbers in the list and divide it by the count of the numbers in the list to come up with the average.

**List arguments**

When you pass a list to a function, the function gets a reference to the list. If the function modifies a list parameter, the caller sees the change. For example, delete\_head removes the first element from a list:

**def** delete\_head(t):

**del** t[0]

Here’s how it is used:

>>> letters = ['a', 'b', 'c']

>>> delete\_head(letters)

>>>print(letters)

['b', 'c']

The parameter t and the variable letters are aliases for the same object.

It is important to distinguish between operations that modify lists and operations that create new lists. For example, the append method modifies a list, but the + operator creates a new list:

>>> t1 = [1, 2]

>>> t2 = t1.append(3)

>>>print(t1)

[1, 2, 3]

>>>print(t2)

None

>>> t3 = t1 + [3]

>>>print(t3)

[1, 2, 3]

>>> t2 is t3

False

This difference is important when you write functions that are supposed to modify lists. For example, this function *does not* delete the head of a list:

**def** bad\_delete\_head(t):

t = t[1:] *# WRONG!*

The slice operator creates a new list and the assignment makes t refer to it, but none of that has any effect on the list that was passed as an argument.

An alternative is to write a function that creates and returns a new list. For example, tail returns all but the first element of a list:

**def** tail(t):

**return** t[1:]

This function leaves the original list unmodified. Here’s how it is used:

>>> letters = ['a', 'b', 'c']

>>> rest = tail(letters)

>>>print(rest)

['b', 'c']

**2.11 Dictionaries**

A *dictionary* is like a list, but more general. In a list, the index positions have to be integers; in a dictionary, the indices can be (almost) any type.

You can think of a dictionary 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.

>>> eng2sp = dict()

>>>print(eng2sp)

{}

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

>>> 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:

>>>print(eng2sp)

{'one': 'uno'}

This output format is also an input format. For example, you can create a new dictionary with three items. But if you print eng2sp, you might be surprised:

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

>>>print(eng2sp)

{'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.

But that’s not a problem because the elements of a dictionary are never indexed with integer indices. Instead, you use the keys to look up the corresponding values:

>>>print(eng2sp['two'])

*'dos'*

The key ’two’ always maps to the value “dos” so the order of the items doesn’t matter.

If the key isn’t in the dictionary, you get an exception:

>>>print(eng2sp['four'])

KeyError: 'four'

The len function works on dictionaries; it returns the number of key-value pairs:

>>>len(eng2sp)

3

The in operator works on dictionaries; it tells you whether something appears as a *key* in the dictionary (appearing as a value is not good enough).

>>>'one' in eng2sp

True

>>>'uno' in eng2sp

False

To see whether something appears as a value in a dictionary, you can use the method values, which returns the values as a list, and then use the in operator:

>>> vals = list(eng2sp.values())

>>>'uno' in vals

True

The in operator uses different algorithms for lists and dictionaries. For lists, it uses a linear search algorithm. As the list gets longer, the search time gets longer in direct proportion to the length of the list. For dictionaries, Python uses an algorithm called a *hash table* 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. I won’t explain why hash functions are so magical, but you can read more about it at wikipedia.org/wiki/Hash\_table.

**2.12 Tuples**

**Tuples are immutable**

A tuple1 is a sequence of values much like a list. The values stored in a tuple can be any type, and they are indexed by integers. The important difference is that tuples are *immutable*. Tuples are also *comparable* and *hashable* so we can sort lists of them and use tuples as key values in Python dictionaries. Syntactically, a tuple is a comma-separated list of values:

>>> t = 'a', 'b', 'c', 'd', 'e'

Although it is not necessary, it is common to enclose tuples in parentheses to help

us quickly identify tuples when we look at Python code:

>>> t = ('a', 'b', 'c', 'd', 'e')

To create a tuple with a single element, you have to include the final comma:

>>> t1 = ('a',)

>>>type(t1)

<type 'tuple'>

Without the comma Python treats (’a’) as an expression with a string in parentheses that evaluates to a string:

>>> t2 = ('a')

>>>type(t2)

<type 'str'>

Another way to construct a tuple is the built-in function tuple. With no argument, it creates an empty tuple:

>>> t = tuple()

>>>print(t)

()

If the argument is a sequence (string, list, or tuple), the result of the call to tuple is a tuple with the elements of the sequence:

>>> t = tuple('lupins')

>>>print(t)

('l', 'u', 'p', 'i', 'n', 's')

Because tuple is the name of a constructor, you should avoid using it as a variable name.

Most list operators also work on tuples. The bracket operator indexes an element:

>>> t = ('a', 'b', 'c', 'd', 'e')

>>>print(t[0])

*'a'*

And the slice operator selects a range of elements.

>>>print(t[1:3])

('b', 'c')

But if you try to modify one of the elements of the tuple, you get an error:

>>> t[0] = 'A'

TypeError: object doesn't support item assignment

You can’t modify the elements of a tuple, but you can replace one tuple with another:

>>> t = ('A',) + t[1:]

>>>print(t)

('A', 'b', 'c', 'd', 'e')

**Comparing tuples**

The comparison operators work with tuples and other sequences. Python starts by comparing the first element from each sequence. If they are equal, it goes on to the next element, and so on, until it finds elements that differ. Subsequent elements are not considered (even if they are really big).

>>> (0, 1, 2) < (0, 3, 4)

True

>>> (0, 1, 2000000) < (0, 3, 4)

True

The sort function works the same way. It sorts primarily by first element, but in the case of a tie, it sorts by second element, and so on. This feature lends itself to a pattern called *DSU* for

**Decorate** a sequence by building a list of tuples with one or more sort keys preceding the elements from the sequence,

**Sort** the list of tuples using the Python built-in sort, and **Undecorate** by extracting the sorted elements of the sequence.

[DSU]

For example, suppose you have a list of words and you want to sort them from longest to shortest:

txt = 'but soft what light in yonder window breaks'

words = txt.split()

t = list()

**for** word in words:

t.append((len(word), word))

t.sort(reverse=True)

res = list()

**for** length, word in t:

res.append(word)

print(res)

*# Code:* [*http://www.py4e.com/code3/soft.py*](http://www.py4e.com/code3/soft.py)

The first loop builds a list of tuples, where each tuple is a word preceded by its length.

sort compares the first element, length, first, and only considers the second element to break ties. The keyword argument reverse=True tells sort to go in decreasing order.

The second loop traverses the list of tuples and builds a list of words in descending order of length. The four-character words are sorted in *reverse* alphabetical order, so “what” appears before “soft” in the following list.

The output of the program is as follows:

['yonder', 'window', 'breaks', 'light', 'what',

'soft', 'but', 'in']

Of course the line loses much of its poetic impact when turned into a Python list and sorted in descending word length order.

**Tuple assignment**

One of the unique syntactic features of the Python language is the ability to have a tuple on the left side of an assignment statement. This allows you to assign more than one variable at a time when the left side is a sequence.

In this example we have a two-element list (which is a sequence) and assign the first and second elements of the sequence to the variables x and y in a single statement.

>>> m = [ 'have', 'fun' ]

>>> x, y = m

>>> x

*'have'*

>>> y

*'fun'*

>>>

It is not magic, Python *roughly* translates the tuple assignment syntax to be the following:2

>>> m = [ 'have', 'fun' ]

>>> x = m[0]

>>> y = m[1]

>>> x

*'have'*

>>> y

*'fun'*

>>>

Stylistically when we use a tuple on the left side of the assignment statement, we omit the parentheses, but the following is an equally valid syntax:

>>> m = [ 'have', 'fun' ]

>>> (x, y) = m

>>> x

*'have'*

>>> y

*'fun'*

>>>

A particularly clever application of tuple assignment allows us to *swap* the values of two variables in a single statement:

>>> a, b = b, a

Both sides of this statement are tuples, but the left side is a tuple of variables; the right side is a tuple of expressions. Each value on the right side is assigned to its respective variable on the left side. All the expressions on the right side are evaluated before any of the assignments.

The number of variables on the left and the number of values on the right must be the same:

>>> a, b = 1, 2, 3

ValueError: too many values to unpack

More generally, the right side can be any kind of sequence (string, list, or tuple).

For example, to split an email address into a user name and a domain, you could write:

>>> addr = 'monty@python.org'

>>> uname, domain = addr.split('@')

The return value from split is a list with two elements; the first element is assigned to uname, the second to domain.

>>>print(uname)

monty

>>>print(domain)

python.org

**2.13 Files**

Inbuilt functions for creating, writing and reading files

Access mode

Access modes govern the type of operations possible in the opened file. It refers to how the file will be used once it’s opened.

1. Write Only (‘w’) : Open the file for writing
2. Write and Read (‘w+’) : Open the file for reading and writing.
3. Append Only (‘a’) : Open the file for writing. The file is created if it does not exist.

**Opening files**

When we want to read or write a file (say on your hard drive), we first must *open* the file. Opening the file communicates with your operating system, which knows where the data for each file is stored. When you open a file, you are asking the operating system to find the file by name and make sure the file exists. In this example, we open the file mbox.txt, which should be stored in the same folder that you are in when you start Python. You can download this file from

[www.py4e.com/code3/mbox.txt](http://www.py4e.com/code3/mbox.txt)

>>> fhand = open('mbox.txt')

>>> print(fhand)

<\_io.TextIOWrapper name='mbox.txt' mode='r' encoding='cp1252'>

If the open is successful, the operating system returns us a *file handle*. The file handle is not the actual data contained in the file, but instead it is a “handle” that we can use to read the data. You are given a handle if the requested file exists and you have the proper permissions to read the file.

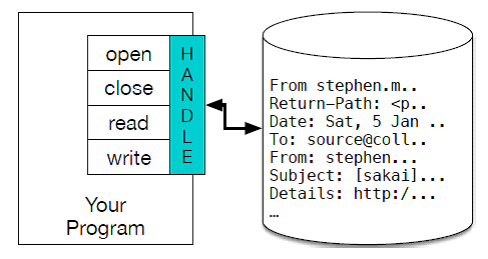


Figure 2.13: A File Handle

If the file does not exist, open will fail with a traceback and you will not get a handle to access the contents of the file:

>>> fhand = open('stuff.txt')

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

FileNotFoundError: [Errno 2] No such file or directory: 'stuff.txt'

Later we will use try and except to deal more gracefully with the situation where we attempt to open a file that does not exist.

**2.14 Reading files**

While the *file handle* does not contain the data for the file, it is quite easy to construct a for loop to read through and count each of the lines in a file:

fhand = open('mbox-short.txt')

count = 0

**for** line in fhand:

count = count + 1

print('Line Count:', count)

*# Code:* [*http://www.py4e.com/code3/open.py*](http://www.py4e.com/code3/open.py)

We can use the file handle as the sequence in our for loop. Our for loop simply counts the number of lines in the file and prints them out. The rough translation of the for loop into English is, “for each line in the file represented by the file handle, add one to the count variable.”

The reason that the open function does not read the entire file is that the file might be quite large with many gigabytes of data. The open statement takes the same amount of time regardless of the size of the file. The for loop actually causes the data to be read from the file.

When the file is read using a for loop in this manner, Python takes care of splitting the data in the file into separate lines using the newline character. Python reads each line through the newline and includes the newline as the last character in the line variable for each iteration of the for loop.

Because the for loop reads the data one line at a time, it can efficiently read and count the lines in very large files without running out of main memory to store the data. The above program can count the lines in any size file using very little memory since each line is read, counted, and then discarded.

If you know the file is relatively small compared to the size of your main memory, you can read the whole file into one string using the read method on the file handle.

>>> fhand = open('mbox-short.txt')

>>> inp = fhand.read()

>>> print(len(inp))

94626

>>> print(inp[:20])

From stephen.marquar

In this example, the entire contents (all 94,626 characters) of the file mbox-short.txt are read directly into the variable inp. We use string slicing to print out the first 20 characters of the string data stored in inp. When the file is read in this manner, all the characters including all of the lines and newline characters are one big string in the variable *inp*. Remember that this form of the open function should only be used if the file data will fit comfortably in the main memory of your computer.

If the file is too large to fit in main memory, you should write your program to read the file in chunks using a for or while loop.

**2.15 Writing Files**

There are two ways to write in a file.

1.write() : Inserts the string str1 in a single line in the text file.

File\_object.write(str1)

2. writelines() : For a list of string elements, each string is inserted in the text file. Used to insert multiple strings at a single time.

File\_object.writelines(L) for L = [str1, str2, str3]

Example Program

# Opening a file

file1 = open('myfile.txt', 'w')

L = ["This is Delhi \n", "This is Paris \n", "This is London \n"]

s = "Hello\n"

# Writing a string to file

file1.write(s)

# Writing multiple strings

# at a time

file1.writelines(L)

# Closing file

file1.close()

# Checking if the data is

# written to file or not

file1 = open('myfile.txt', 'r')

print(file1.read())

file1.close()

**Output**

Hello

This is Delhi

This is Paris

This is London

2.15 **Classes**

A class is a user-defined data type which includes local methods and properties.One of the big differences between functions and classes is that a class contains both data (variables) called properties and methods (functions defined inside a class).

An Object is an instance of a Class. A class is like a blueprint while an instance is a copy of the class with actual value

Syntax - Class classname :

//Body of the class

Example -

class Shape:

sides = 4 #first property

name = "Square" #second property

def description(a): #method defined

return ("A square with 4 sides")

s1 = Shape() #creating an object of Shape

print "Name of shape is:",(s1.name)

print "Number of sides are:",(s1.sides)

print s1.description()

**2.16 Constructors**

Constructors are generally used for instantiating an object. The task of constructors is to initialize(assign values) to the data members of the class when an object of the class is created. In Python the \_\_init\_\_() method is called the constructor and is always called when an object is created.

Syntax of constructor declaration :

def \_\_init\_\_(self):

# body of the constructor

Constructor Types

1. **default constructor**: The default constructor is a simple constructor which doesn’t accept any arguments. Its definition has only one argument which is a reference to the instance being constructed.

2. **parameterized constructor**: constructor with parameters is known as parameterized constructor. The parameterized constructor takes its first argument as a reference to the instance being constructed known as self and the rest of the arguments are provided by the programmer.

class GeekforGeeks:

**Example of the Default Constructor**

# default constructor

def \_\_init\_\_(self):

self.geek = "GeekforGeeks"

# a method for printing data members

def print\_Geek(self):

print(self.geek)

# creating object of the class

obj = GeekforGeeks()

# calling the instance method using the object obj

obj.print\_Geek()

Output **- GeekforGeeks**

**Example of the parameterized constructor :**

class Addition:

first = 0

second = 0

answer = 0

# parameterized constructor

def \_\_init\_\_(self, f, s):

self.first = f

self.second = s

def display(self):

print("First number = " + str(self.first))

print("Second number = " + str(self.second))

print("Addition of two numbers = " + str(self.answer))

def calculate(self):

self.answer = self.first + self.second

# creating object of the class

# this will invoke parameterized constructor

obj = Addition(1000, 2000)

# perform Addition

obj.calculate()

# display result

obj.display()

**Output**

First number = 1000

Second number = 2000

Addition of two numbers = 3000

2.17 **Class Inheritance**

Inheritance is a way by which a subclass can inherit the attributes and methods of another class. The new class is called derived (or child) class and the one from which it inherits is called the base (or parent) class

Types of Inheritance

1.**Single inheritance** - the derived class is derived from only one base class.

**Syntax** -

class BaseClass:

Base class body

class DerivedClass(BaseClass):

Derived class body

2.**Multiple Inheritance -** In multiple inheritance, the derived class is derived from more than one base class

**Syntax** -

class BaseClass1:

Base class1 body

class BaseClass:

Base class2 body

class DerivedClass(BaseClass1,BaseClass2):

Derived class body

3. **Multilevel Inheritance -** In multilevel inheritance, the derived class is derived from another derived class.

**Syntax** -

class Base:

Base Class body

class Derived1(Base):

Derived Class1 body

class Derived2(Derived1):

Derived Class2 body

**Benefits of Class Inheritance**

1.Reusability of code - Since the derived class inherits features from the base class, adding new features to it. This results in re-usability of code. This makes the code more scalable.

2. Structured Code - By dividing the code into classes, we can structure our software better by dividing functionality into classes.

2.18 **Overloadiong**

**1.Operator Overloading**

**2.Function Overloading**

**Operator Overloading**

Operator Overloading means giving extended meaning beyond their predefined operational meaning. For example operator + is used to add two integers as well as join two strings and merge two lists. It is achievable because ‘+’ operator is overloaded by int class and str class. You might have noticed that the same built-in operator or function shows different behavior for objects of different classes, this is called Operator Overloading.

Overloading + Operator

# Python Program illustrate how

# to overload an binary + operator

class A:

def \_\_init\_\_(self, a):

self.a = a

# adding two objects

def \_\_add\_\_(self, o):

return self.a + o.a

ob1 = A(1)

ob2 = A(2)

ob3 = A("Geeks")

ob4 = A("For")

print(ob1 + ob2)

print(ob3 + ob4)

**Output**

3

GeeksFor

**Method Overloading**

Python does not support method overloading by default. But there are different ways to achieve method overloading in Python.The problem with method overloading in Python is that we may overload the methods but can only use the latest defined method.

Example 1 - Where it uses the latest method only

# First product method.

# Takes two argument and print their

# product

def product(a, b):

p = a \* b

print(p)

# Second product method

# Takes three argument and print their

# product

def product(a, b, c):

p = a \* b\*c

print(p)

# Uncommenting the below line shows an error

# product(4, 5)

# This line will call the second product method

product(4, 5, 5)

Achieving Method Overloading using Multiple Dispatch Decorator

from multipledispatch import dispatch

#passing one parameter

@dispatch(int,int)

def product(first,second):

result = first\*second

print(result);

#passing two parameters

@dispatch(int,int,int)

def product(first,second,third):

result = first \* second \* third

print(result);

#you can also pass data type of any value as per requirement

@dispatch(float,float,float)

def product(first,second,third):

result = first \* second \* third

print(result);

#calling product method with 2 arguments

product(2,3,2) #this will give output of 12

product(2.2,3.4,2.3) # this will give output of 17.985999999999997

**Output** -

12

17.985999999999997

In Backend, Dispatcher creates an object which stores different implementation and on runtime, it selects the appropriate method as the type and number of parameters passed.