Contents

Part I

Python Basics

Chapter 1

Python Basics

1.1 Gradual Typing System

Gradual typing:

- allows one to annotate only part of a program, thus leverage desirable aspects of both dynamic and static typing
- uses new relationship: is-consistent-with to describe the connection between two types. This is similar to is-subtype-of
- uses a type: `Any`. `Any` is a special type that indicates an unconstrained type.

The is-consistent-with relationship is defined by three rules:

- A type `type_1` is said "is consistent with" another type `type_2`, if `type_1` is a subtype of `type_2` (but not the other way around)
- `Any` is defined to be consistent with every type (but `Any` is not a subtype of every type).
- Every type is defined to be consistent with `Any` (but every type is not a subtype of `Any`).

Chapter 2

Operators

2.1 ^

Bitwise XOR. It takes two bit patterns of equal length and performs the logical exclusive OR operation on each pair of corresponding bits. The result in each position is 1 if only the first bit is 1 or only the second bit is 1, but will be 0 if both are 0 or both are 1.

In another word, for two bit patterns of equal length, we perform the comparison of two bits at the same position, being 1 if the two bits are different, and 0 if they are the same.

Example:

```
101 (decimal 5) XOR
011 (decimal 3) =
110 (decimal 6)
```

Run in python:

```
1 a = 5
2 b = 3
3 print(bin(a ^ b))
```

Output:

```
0b110
```

Chapter 3

Data Types

3.1 General Idea

3.1.1 Variable Declaration

When you declare a variable with some initializing value:

```
character_name = "Mike"
```

You don't have to define the type of variable. The variable will be the type that following the assignment operator. For example, in the above code, the type of `character_name` is string. Another point derived from this is, you can use a variable to store another data type value. For example:

```
character_name = "Mike"
print(character_name)
character_name = 123
print(character_name) + 3
```

The result would be:

```
Mike
126
```

3.1.2 Python Names and Values

Ned Batchelder has a very neat introduction on the basic behavior of python's names and values. Check it out first. A video presentation is here.

3.1.2.1 Example 1: try to change value refered in list

```
1 nums = [1, 2, 3]
2 for num in nums[:3]:
3     num += 1
4     print(num)
5 print(nums)
```

Output:

```
2
3
4
[1, 2, 3]
```

In this example, `num` in each iteration is first referred to the value where the list element is referring to. However, the assignment step will create a new value and refer to `num`, not the same value referred by list element. The process can be shown below:

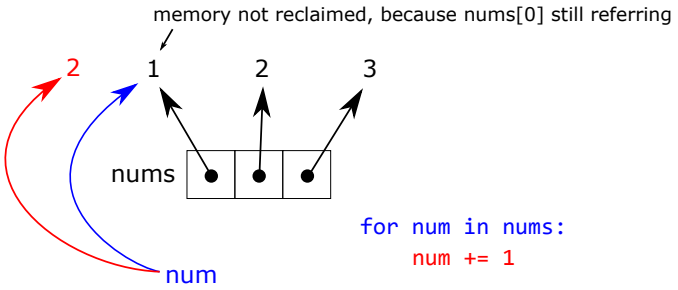

Figure 3.1: Try to change value referenced in list

3.1.2.2 Example 2

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

```
for num in nums:
    num[0] += 1
    print(num)
print(nums)
```

Output:

```
[2]
[3]
[4]
[[2], [3], [4]]
```

The process can be shown below:

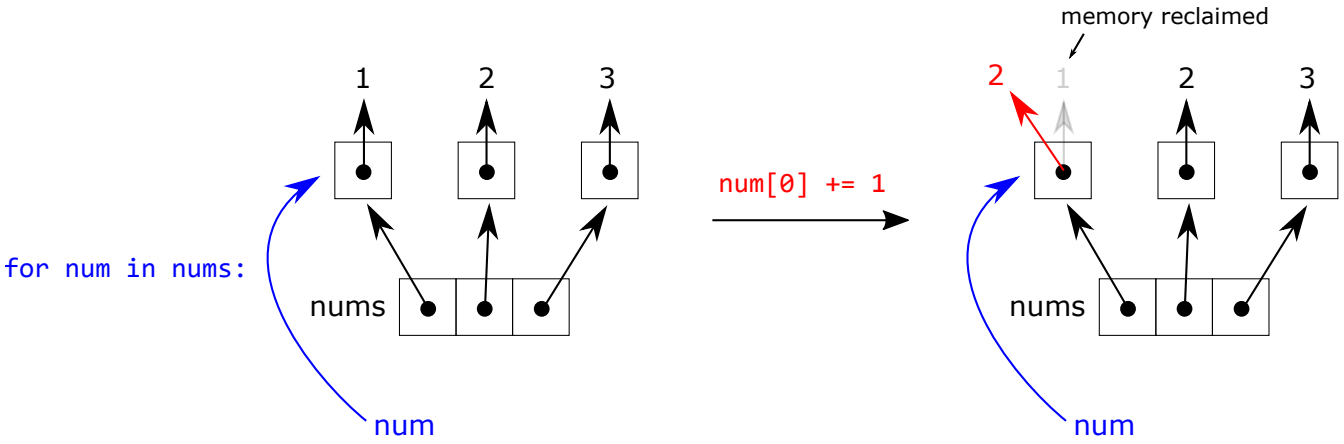

Figure 3.2: Try to change value referenced in list of list

An unreferenced value will be reclaimed by python. To demonstrate this, we can use another variable to reference one of the value originally referenced by the list of list:

```
nums = [[1], [2], [3]]
first_num = nums[0][0]   # keep reference

for num in nums:
    num[0] += 1
```

```
6       print(num)
7   print(nums)
8
9   print(first_num)  # check original value's existence
```

Output:

```
[2]
[3]
[4]
[[2], [3], [4]]
1
```

The process is shown below:

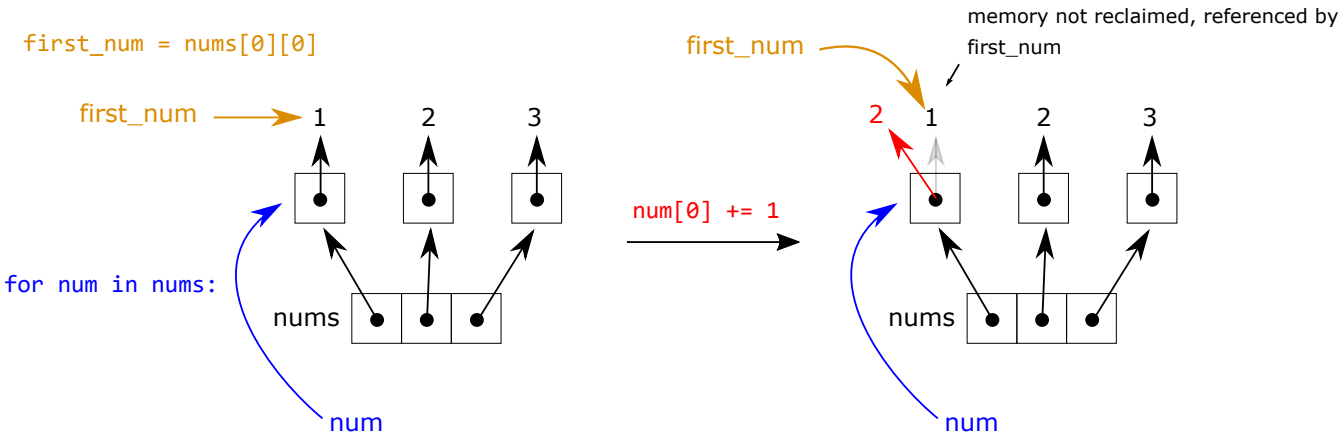

Figure 3.3: Try to keep value from being reclaimed

3.1.3 Type Annotations

Python is a gradual typing system. It allows not only dynamical typing but also static typing. It uses type annotations (also known as type hints) to allows one to specify types of variables and function arguments/return values. The syntax is as follows:

```
my_variable: int # variable annotation

# function annotation
def greeting(name: str) -> str:
    return 'Hello ' + name
```

The `-> str` part has annotated the return type of the function.

However, you can still assign different types to a variable that is annoted otherwise. Python will still run (although in pycharm, IDE will notify you of the unmatched type issue).

3.1.4 Pylint

Pylint is a python source code analyzer that helps you maintain good coding standards.

3.2 Bool Value

In Python, the two bool values are:

```
True
False
```

Notice that, the first letter is capitalized. Unlike in C++ (`true, false`).

3.3 String

String is a sequence of characters in python. String is immutable. The python string documentation can be found here.

3.3.1 String Formating

Placing 'r' before a string will yield its raw value. For example:

```
print(r"\n\n\n")
```

Output:

```
\n\n\n
```

3.3.2 Access Single Character by []

Just like string in C++, you can access the individual character in a string by using `[]`. For example:

```
content = "abcde"
print(content[0])
```

Output:

```
a
```

The indexing of string also supports negative index. Check List section in this notes for more details. For example:

```
content = "abcde"
print(content[-1])
print(content[-3])
```

Output:

```
e
c
```

One key difference between python string and C++ string is that, you can't modify the single character in string by the bracket operator. For example:

```
string = "abcde"
string[0] = 'a'
```

Output:

```
TypeError: 'str' object does not support item assignment
```

3.3.3 Functions

3.3.3.1 General Idea

Just like string class in C++, a data type in python has member functions. This is even true for primitive types in C++ (for example, `int, double`), i.e. these "primitive" types also have their own member functions.

To call the member function in python's data type, you use `.` operator (same as C++):

```
1 content = "abc"
2 print(content.isupper())
```

In the above code, the function `isupper()` will return a Boolean value indicating whether the string variable contains all upper case letters. In this case, the result would be:

```
False
```

You can concatenate the calling of member functions. For example:

```
1 content = "abc123"
2 print(content.upper().isupper())
3 print(content)
```

In line 2, the member function `upper()` of the string variable `content` is called first. This function will return a string variable which contains the copy of `content` variable, except that all lower case letters are transformed to upper case letters. Then, the member function `isupper()` of this **RETURNED** variable will be called to check if all letters in this **RETURNED** variable are upper case. After this, in line 3, `content` will be printed again to show the data stored in `content` hasn't been changed by the `upper()` member function. The output is:

```
True
abc123
```

Some often used member functions and global functions are listed in the following sections.

3.3.3.2 `float()`

This global function will convert a string into a float number, and will return this number. The input string must be only in the format of a float number, i.e. either x or x.x, where x can be only digit, otherwise, errors will be thrown.

Example:

```
num = input("Enter a number: ")
print(float(num))
```

Output:

```
Enter a number; 5.5
5.5
```

3.3.3.3 `int()`

This global function will convert a string into an integer number, and will return this number. The input string must contain only digit, otherwise, errors will be thrown.

Example:

```
num = input("Enter a number: ")
print(int(num))
```

Output:

```
Enter a number; 5
5
```

3.3.3.4 `len()`

This function will return an integer that indicates the length of a string variable `string`. It is **not** a member function of string, but a global function, which accepts "sized" parameter in python (iterable object).

3.3.3.5 .count()

This function accepts a string, it will return an integer that indicates the number of times the string appeared in the calling string. Example:

```
string = "abcdede"
print("Counting result : ", string.count("de"))
```

Output:

```
Counting result :  2
```

3.3.3.6 .index()

This member function will return the starting index value of a character or sub-string in the string (first appearing position). For example:

```
string = "abcd 123 apple"
print(string.index(" "))
print(string.index(" ap"))
```

Output:

```
4
8
```

When the parameter (character or substring) is not in the string, an error will be thrown:

```
string = "abcd 123 apple"
print(string.index("pee"))
```

Output:

```
ValueError: substring not found
```

The code lines that are after this line will not be executed, program will stop.

3.3.3.7 `.islower()`

This function will return a Boolean value to check if **ALL** letters in `string` are lowercase. Its a member function of string.

3.3.3.8 `.isupper()`

This function will return a Boolean value to check if **ALL** letters in `string` are uppercase. Its a member function of string.

3.3.3.9 `.join()`

This function accepts an iterable sequence object. It will return a string which is built using following logic: concatenate iterated item in the sequence and the original string to make a combined string, this is done for each of the item in the sequence. Then return the combination of all these strings. For example, if the sequence is `['1', '2', '3']`, and the calling string is: `"abc"`, then following substrings will be first made;

- `"1abc"`
- `"2abc"`
- `"3abc"`

Then they will be combined to gether to give: `"1abc2abc3abc"`.

Example:

```
t1 = ('c', 't', 'y')
string = "123"
print(string.join(t1))
```

Output:

```
c123t123y
```

This function can be used to convert a list to a string, just choose an empty string to call the `.join()` function:

```
t1 = ('c', 't', 'y')
print("".join(t1))
```

3.3.3.10 .lower()

This function will return a copy of `string` with all letters lower cased. It is useful for storing data: transform your string data into lower case so you can manage them with ease.

3.3.3.11 .replace()

This function accepts two string parameters:

```
string.replace("search_key", "replace_key")
```

It will search for the existence of `"search_key"` in the string. If the key being found, it will be replaced by `"replace_key"`. This "replaced" version of the string will be returned by this function (as a copy of the original string). For example:

```
content = "accab"
print(content.replace("a", "c"))
print(content)
```

Output:

```
ccccb
accab
```

When no match found, no error will be thrown. Code lines followed will be executed.

3.3.3.12 .title()

Change the first letter of each word into upper case. Example:

```
string = "this is a title"
print(string.title())
```

Output:

```
This Is A Title
```

3.3.3.13 `.rstrip()` and `.lstrip()`

This function will return a string that stripped any whitespaces on the right or left side of the `self` string. For example:

```
1 string = ["\tthis is a title    "]
2 print("Before: ", string)
3 print("After .rstrip(): ", [string[0].rstrip()])
4 print("After .lstrip(): ", [string[0].lstrip()])
```

Output:

```
Before:  ['\tthis is a title    ']
After .rstrip():  ['\tthis is a title']
After .lstrip():  ['this is a title    ']
After both:  ['this is a title']
```

To strip the whitespaces at right end and left end simultaneously, you can use `.strip()`. In the real world, these stripping functions are used most often to clean up user input before it's stored in a program.

3.3.3.14 `.split()`

This method separates the string into substrings wherever it finds a space and stores all the substrings in a list. For example:

```
1 title = "Alice in Wonderland!"
2 print(title.split())
```

Output:

```
['Alice', 'in', 'Wonderland!']
```

3.3.3.15 .upper()

This function will return a copy of `string` with all letters upperized. Its a member function of string.

3.3.4 Chinese Support in Python String

Unlike C++'s string class which needs to specify different encoding system in order to hold Chinese characters, you can directly assign Chinese character to a python variable.

3.4 Numerics

The subtypes are int, float and complex.

3.4.1 Functions

3.4.1.1 str(num)

You can use this global function to convert a number variable to a string object. It will return the string version of the number variable. For example:

```
1 num = 5.45
2 print(str(num).index("."))
```

Output:

```
1
```

3.4.1.2 abs(num)

This global function will return a copy of the abusolute value of `num`. Also, there is a member function can do the same thing:

```
1 num = -5.35
2 print(abs(num))
3 print(num.__abs__())
```

Output:

```
5.35
5.35
```

3.4.1.3 `pow()`

A global function that accepts two parameters: `pow(base, exponent)`. It will return the result.

```
print(pow(2, 3))
```

Output:

```
8
```

3.4.1.4 `max()`

This global function will return a copy of the maximum numbers in the passed in parameter. For example:

```
print(max(2, -1, 3, 5, 6, 1, 2, 10))
```

Output:

```
10
```

3.4.1.5 `min()`

This global function will return a copy of the minimum numbers in the passed in parameter. For example:

```
print(min(2, -1, 3, 5, 6, 1, 2, 10))
```

Output:

```
10
```

3.4.1.6 round()

This global function will return a copy of the closest integer of the number passed in. For example:

```
print(round(2.36))
print(round(-2.45))
print(round(-3.1))
```

Output:

```
2
-2
-3
```

3.4.1.7 math::floor()

This function is defined in `math` module. It will return the integer that is no larger than the input parameter. For example:

```
1 from math import *
2 print(floor(2.4))
```

Output:

```
2
```

3.4.1.8 math::ceil()

This function is defined in `math` module. It will return the integer that is no smaller than the input parameter. For example:

```
1 from math import *
2 print(ceil(2.4))
```

Output:

```
3
```

3.4.1.9 `math::sqrt()`

This function is defined in `math` module. It will return the square root of the input parameter.

For example:

```
1 from math import *
2 print(sqrt(4))
```

Output:

2

Chapter 4

Data Structures

4.1 Sequence

4.1.1 General Idea

Sequence is not a specific type of data structure. It is the iterable type, which means it is composed of smaller pieces of information, and these information are stored inside a single entity (sequence).

There are seven sequence subtypes:

- strings
- lists
- tuples
- sets
- bytes
- bytearrays
- range objects

The range data type finds common use in the construction of enumeration-controlled loops.

4.1.2 Common Sequence Operations (for both immutable and mutable)

4.1.2.1 `x in s`

`True` if an item of is is equal to `x`, else `False`

4.1.2.2 `x not in s`

`False` if an item of `s` is equal to `x`, else `True`

4.1.2.3 `s + t`

The concatenation of `s` and `t`

4.1.2.4 `s * n, n * s`

`n` shallow copies of `s` concatenated

4.1.2.5 `s[i]`

`i` th items of `s`, start from 0

4.1.2.6 `s[i:j]`

Slice of `s` from `i` to `j`

4.1.2.7 `s[i:j:k]`

Slice of `s` from `i` to `j` with step `k`

4.1.2.8 `len(s)`

Length of `s`

4.1.2.9 `min(s)`

Smallest item of `s`

4.1.2.10 `max(s)`

Largest item of `s`

4.1.2.11 `s.index(x)`

Index of the first occurrence of `x` in `s`. If `x` not found in `s`, a `ValueError` will be thrown (`ValueError: substring not found`).

4.1.2.12 `s.count(x)`

Total number of occurrences of `x` in `s`. If `x` not found in `s`, no error thrown, just return 0.

4.1.3 Mutable Sequence Operations

4.1.3.1 `s[i] = x`

Item `s[i]` of `s` is replaced by `x`. Here, `x` will be evaluated as an object. If `x` is a sequence, it will replace `s[i]` as a whole. For example:

```
1 list1 = [x for x in range(3, 15, 2)]
2 list2 = [2, 4, 6, 8, 10]
3
4 print(list1)
5 list1[1] = list2
6 print(list1)
```

Output:

```
[3, 5, 7, 9, 11, 13]
[3, [2, 4, 6, 8, 10], 7, 9, 11, 13]
```

4.1.3.2 `s[i:j] = []`

Delete slice of `s` from `i` to `j` (not including `s[j]`). For example:

```
list1 = [x for x in range(3, 15, 2)]
print(list1)
list1[:4] = []
print(list1)
```

Output:

```
[3, 5, 7, 9, 11, 13]
[11, 13]
```

4.1.3.3 `s[i:j] = t`

Slice of `s` from `i` to `j` (not including `s[j]`) is replaced by the contents of `t`, `len(s[i:j])` and `len(t)` does not need to be equal. Here, `t` will be evaluated as a sequence. This statement will produce same result as if you go following steps:

- call `s[i:j] = []` to delete slice of `s` from `i` to `j`
- for each iterated item in `t`, insert them starting from `i`

Example:

```
list1 = [x for x in range(3, 15, 2)]
dic1 = {
    'a': "A",
    'b': "B",
    'c': "C"
}

print(list1)
list1[:4] = dic1
print(list1)
```

Output:

```
[3, 5, 7, 9, 11, 13]
```

```
['a', 'b', 'c', 11, 13]
```

Pay attention that, in a set of key-value pair (i.e. dictionary), only key will be inserted. Also notice that, Slice `list1[:4]` has been deleted, and only three entries added, the size of `list1` is reduced by 1.

Another example showing that `t` is evaluated as an iterable sequence rather than a whole object:

```
1 list1 = [x for x in range(3, 15, 2)]
2 str1 = 'abcdef'
3
4 print(list1)
5 list1[:4] = str1
6 print(list1)
```

Output:

```
[3, 5, 7, 9, 11, 13]
['a', 'b', 'c', 'd', 'e', 'f', 11, 13]
```

So `str1` is inserted one character at a time (each iterated item is a character).

4.1.3.4 `s[i:j:k] = t`

The elements of `s[i:j:k] are replaced by those of ~t`. Pay attention that the number of elements in slice `s[i:j:k]` should be equal to number of elements in `t`. Otherwise, `ValueError` will be thrown (attempt to assign sequence of size `len(t)` to extended slice of size `len(s[i:j:k])`).

Example:

```
1 list1 = [x for x in range(3, 18, 2)]
2 print(list1)
3 list1[:5:2] = [0, 0, 0]
4 print(list1)
```

Output:

```
[3, 5, 7, 9, 11, 13, 15, 17]
[0, 5, 0, 9, 0, 13, 15, 17]
```

4.1.3.5 `del s[i:j]` and `del s[i:j:k]`

Delete the slice in `s` indicated by `[i:j]` and `[i:j:k]`.

4.1.3.6 `s.append(x)`

Add `x` to the end of `s`, here `x` will be evaluated as an object, not a sequence. It will be appended to `s` as a whole. Example:

```
1 list1 = [2, 4, 6, 8, 10]
2 str1 = 'abcdef'
3 print(list1)
4 list1.append(str1)
5 print(list1)
```

Output:

```
[2, 4, 6, 8, 10]
[2, 4, 6, 8, 10, 'abcdef']
```

4.1.3.7 `s.extend(x)`

Append the content of `x` to end of `s`. Here, `x` will be evaluated as a iterable sequence. It is doing the same thing as:

```
1 for item in x:
2     s.append(item)
```

Example:

```
1 list1 = [2, 4, 6, 8, 10]
2 str1 = 'abcdef'
```

```
print(list1)
list1.extend(str1)
print(list1)
```

Output:

```
[2, 4, 6, 8, 10]
[2, 4, 6, 8, 10, 'a', 'b', 'c', 'd', 'e', 'f']
```

4.1.3.8 s.count(x)

Return number of occurrence of `x` in `s`.

4.1.3.9 s.insert(i, x)

Insert `x` at position `i`.

4.1.3.10 s.pop()

This function will remove the last item in the sequence (default operation, when no parameters passed in). Example:

```
name = ["what", "a", "big", "apple"]
print("Before pop: ", name)
name.pop()
print("After pop: ", name)
```

Output:

```
Before clear:  ['what', 'a', 'big', 'apple']
After clear:  ['what', 'a', 'big']
```

This function will return the popped item, you can declare a variable to hold it and do more stuff with it:

```
popped_item = items.pop()
```

When an integer value `i` is passed in, the item at that index will be popped (and returned):

```
items.pop(1)
```

The second item in the list will be popped.

4.1.3.11 s.remove()

Same as `del s[s.index(x)]`. If `x` is not in the list, `ValueError` will be thrown.

4.1.3.12 s.reverse()

Reverses the items of `s` in place

4.1.3.13 s.sort()

This function will sort the sequence in ascending order. For example:

```
nums = [3, 2, 7, 1, 5]
nums.sort()
print(nums)
```

Output:

```
[1, 2, 3, 5, 7]
```

It will use a compare function to determine the order of the item. The comparator should define ordering information among any two pairs of the elements stored in the list. If not, an error will be thrown. For example:

```
name = ["what", "a", "big", "Apple", 2]
name.sort()
print(name)
```

Output:

```
TypeError: '<' not supported between instances of 'int' and 'str'
```

To sort the items reversely (in descending order), you pass `reverse = True` into the `sort()` function. For example:

```
nums = [1, 2, 3, 4, 5]
nums.sort(reverse=True)
print(nums)
```

Output:

```
[5, 4, 3, 2, 1]
```

4.2 List

List is a way of managing data in python. Because a list usually contains more than one element, it's a good idea to make the name of your list plural, such as letters, digits, or names.

4.2.1 Declaration

4.2.1.1 Declare by Constructed List

To declare a list:

```
friend = ["A", "B", "C"]
```

The `[]` will indicate a list structure. So basically the computer knows that a bunch of values will be stored in `friend`.

Unlike arrays in C++, you can put different types into a same list, for example:

```
friend = ["A", "B", 2, 2.5, "C"]
print(friend[1])
print(friend[-2])
```

Output:

```
B
2.5
```

You can print the whole list by passing the name of the list into `print()`, for example:

```
1 friend = ["A", "B", 2, 2.5, "C"]
2 print(friend)
```

Output:

```
['A', 'B', 2, 2.5, 'C']
```

Notice that for non-string element, there is no ' (which indicates string object).

4.2.1.2 Declare by another List

Suppose you have two lists `A` and `B`, and `B` is declared using `=` directly:

```
1 A = [1, 2, 3, 4, 5]
2 B = A
```

In python, line 2 is actually defining a left-reference of `A`, named `B`. They are referencing to the same memory space that hold `[1, 2, 3, 4, 5]`. So, for example, if `A` is reversed, then print `B`, it is also reversed:

```
1 A.reverse()
2 print(B)
```

Output:

```
[5, 1, 7, 2, 3]
```

To copy the content in `A` to `B`, you need to use the `.copy()` member function.

4.2.2 `operator[]`

The specific element can be accessed by the `[]` operator. The indexing is similar with C++, from `0` to `size - 1`. However, python also supports indexing from `-1` to `-size`, the position these negative indexes `i` refered to is the same as `size + i`, so `-1` is the last element, while `-size` is the first element.

You can select portion of the list by using `:`. When you type `array[a:b]`, it will return a subarray, which is portion of the original `array`, where the subarray contains elements in

range `[a,b)`, notice: not including the ending element. For example:

```
nums = [1, 2, 3, 4, 5, "end"]
print(nums[2:4])
```

Output: [3, 4]

4.2.3 `in` keyword

You can use `in` to determine whether an element is inside the list or not. For example:

```
nums = [1, 1, 2, 2, 1, 3, 4, 6, 7, 9]
print(1 in nums)
```

Output:

```
True
```

This expression can be used in many situations, like `if` statement, `while` loop (looping condition).

4.2.4 Slice

You can use slice to get a copy of portions of the list by using colon. For example: `array[a:]`, this will return a sublist that begin at `a`, all the way to the last element of the array. For example:

```
nums = [1, 2, 3, 4, 5, "end"]
print(nums[2:])
```

Output:

```
[3, 4, 5, 'end']
```

If you omit the first index in a slice, python automatically starts your slice at the beginning of the list. For example:

```
nums = [i for i in range(1, 10)]
print(nums[:5])
```

Output:

```
[1, 2, 3, 4, 5]
```

Slice also works for negative index. If left of the slice is omitted, it means from start to where the negative index refer to. If right of the slice is omitted, it means from where the negative index refer to, to the end. Example:

```
1 nums = [i for i in range(1, 10)]
2 print(nums[:-3])
3 print(nums[-3:])
```

Output:

```
[1, 2, 3, 4, 5, 6]
[7, 8, 9]
```

Slice can be used in for loop (as the iterated object). For example, to loop through the first three element in a list:

```
1 nums = [i for i in range(1, 10)]
2 for num in nums[:3]:
3     print(num)
```

Output:

```
1
2
3
```

4.2.5 List comprehension

A list comprehension combines the `for` loop and the creation of new elements into one line, so you can make a list in a simple way. The syntax is as follows:

```
1 nums = [i*i for i in range(1, 10)]
```

Output:

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

4.2.6 2D List

The idea of 2D list is simple. It is a list, whose element object is another list. For example:

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9],
    [0]
]
```

Each list object is a row in the 2D matrix. Each object in each row list is the column of the 2D matrix. To access the specific element, we use double bracket operator. For example, assume the above matrix is declared, then `matrix[0][2] == 3`.

Example: use a nested for loop to traverse the 2D lists defined above. We have two ways, the first one is to use bracket operator:

```
for row in range(len(matrix)):
    for column in range(len(matrix[row])):
        print(matrix[row][column])
```

Output:

```
1
2
3
4
5
6
7
8
9
```

```
0
```

The second way is to use range based idea:

```
for row in matrix:
    for col in row:
        print(col)
```

Output:

```
1
2
3
4
5
6
7
8
9
0
```

4.2.7 Functions

4.2.7.1 `.extend()`

This is a member function of list. It requires two parameters: `self` and `iterable`. `self` is the list object that are calling this function, you generally don't have to pass in again. `iterable` is an object that is, iterable. For example, a string is iterable in the sense that it consists of multiple components (the characters). Another list is also iterable.

This function will extend the `self` list by adding the each iterated item in the `iterable`. For example:

```
nums = [1, 2, 3, 4, 5]
name = ["what", "a", "big", "apple"]
```

```
nums.extend(name)
print(nums)
```

Output:

```
[1, 2, 3, 4, 5, 'what', 'a', 'big', 'apple']
```

Pay attention to the idea of "each iterated item". If we extend a list with a string:

```
string = "abcde"
nums = [1, 2, 3, 4, 5]
nums.extend(string)
print(nums)
```

Output will be:

```
[1, 2, 3, 4, 5, 'a', 'b', 'c', 'd', 'e']
```

This is because in string, each character is stored in one slot, i.e. each iterated item is a character, rather than the whole string. We can, however, to construct a list with just one string element, and extend `self` with this **LIST** (rather than the string):

```
string = "abcde"
nums = [1, 2, 3, 4, 5]
nums.extend([string])
print(nums)
```

Output:

```
[1, 2, 3, 4, 5, 'abcde']
```

here, the string is first used to construct a list, and `nums` is extended using this list.

Another example:

```
nums = [1, 2, 3, 4, 5]
name = ["what", "a", "big", "apple"]
nums.extend(name[2])
```

```
print(nums)
```

Output:

```
[1, 2, 3, 4, 5, 'b', 'i', 'g']
```

The thing is, whatever you put in `.extend()`, python will interpret it as an `iterable`, not an actual object. So you can't expect the above code produces result like: `[1, 2, 3, 4, 5, 'big']`.

4.2.7.2 `.append()`

Unlike `.extend()`, you can use `.append()` to append an object to the calling list. Whatever you put in `.append()`, it will be interpreted as an object, not an `iterable`. For example:

```
string = "abcde"
nums = [1, 2, 3, 4, 5]
name = ["what", "a", "big", "apple"]
nums.append(string)
nums.append(name[2])
print(nums)
```

Output:

```
[1, 2, 3, 4, 5, 'abcde', 'big']
```

If you pass name of another list into it, you will have nested list structure, i.e. one element of the list is another list. For example:

```
string = "abcde"
nums = [1, 2, 3, 4, 5]
name = ["what", "a", "big", "apple"]
nums.append(string)
nums.append(name)
print(nums)
```

Output:

```
[1, 2, 3, 4, 5, 'abcde', ['what', 'a', 'big', 'apple']]
```

4.2.7.3 .insert()

This member function can insert a data item into certain position in a list. It accepts three parameters: `self`, `index` and `object`.

- `self` is always the calling list (so you don't have to input).
- `index` is the position you want to insert the data item into. The item will be inserted to the **FRONT** of the `list[index]`. You can provide an index that is beyond the range of the calling list. If its too large, item will be inserted to the back. If its too small (e.g. smaller than `-size`), it will be inserted to the front.
- `object` is the data item you want to insert

For example:

```
1 string = "abcde"
2 nums = [1, 2, 3, 4, 5]
3 name = ["what", "a", "big", "apple"]
4 nums.insert(-7, name)
5 print(nums)
```

Output:

```
[['what', 'a', 'big', 'apple'], 1, 2, 3, 4, 5]
```

Elements after the insertion position are shifted one position toward the end (like C++ vector?)

4.2.7.4 .remove()

This member function can remove certain element in the list. It accepts two parameters: `self` and `object`.

- `self` is always the calling list (so you don't have to input).

- `object` is the item you wish to remove in the list

For example:

```
nums = [1, 2, 3, 4, 5]
name = ["what", "a", "big", "apple"]
nums.insert(3, name)
print("Before removing: ", nums)
nums.remove(name)
print("After removing: ", nums)
```

Output:

```
Before removing:  [1, 2, 3, ['what', 'a', 'big', 'apple'], 4, 5]
After removing:  [1, 2, 3, 4, 5]
```

Pay attention that if there the `object` is not found in the list, an error will be thrown.

4.2.7.5 .pop()

This function will remove the last item in the list (default operation, when no parameters passed in). Example:

```
name = ["what", "a", "big", "apple"]
print("Before pop: ", name)
name.pop()
print("After pop: ", name)
```

Output:

```
Before clear:  ['what', 'a', 'big', 'apple']
After clear:  ['what', 'a', 'big']
```

This function will return the popped item, you can declare a variable to hold it and do more stuff with it:

```
popped_item = items.pop()
```

When an integer value `i` is passed in, the item at that index will be popped (and returned):

```
items.pop(1)
```

The second item in the list will be popped.

4.2.7.6 `del`

`del` is not a function, but it is a statement that can remove element in list. you can remove the element with index `i` using following line:

```
del items[i]
```

4.2.7.7 `.clear()`

This function will clear the content of the list. Example:

```
name = ["what", "a", "big", "apple"]
print("Before clear: ", name)
name.clear()
print("After clear: ", name)
```

Output:

```
Before clear:  ['what', 'a', 'big', 'apple']
After clear:  []
```

4.2.7.8 `.index()`

This function accepts an object, and will try to search if there is any match in the calling list. If so, it will return the index of the object, otherwise, an error will be thrown (the same as string's `index()` member function). Example:

```
name = ["what", "a", "big", "apple"]
print("The index : ", name.index("apple"))
```

Output:

```
The index :  3
```

4.2.7.9 .count()

This function accepts an object. It will return an integer showning the number of appearing times in the calling list. Example:

```
name = ["what", "a", "big", "apple", 2]
print("Counting result : ", name.count(2))
```

Output:

```
Counting result :  1
```

The object type allowed by this function is determined by the type of object the list currently have.

4.2.7.10 .sort()

This function will sort the list in ascending order. For example:

```
nums = [3, 2, 7, 1, 5]
nums.sort()
print(nums)
```

Output:

```
[1, 2, 3, 5, 7]
```

It will use a compare function to determine the order of the item. The comparator should define ordering information among any two pairs of the elements stored in the list. If not, an error will be thrown. For example:

```
name = ["what", "a", "big", "Apple", 2]
name.sort()
print(name)
```

Output:

```
TypeError: '<' not supported between instances of 'int' and 'str'
```

To sort the items reversely (in descending order), you pass `reverse = True` into the `sort()` function. For example:

```
nums = [1, 2, 3, 4, 5]
nums.sort(reverse=True)
print(nums)
```

Output:

```
[5, 4, 3, 2, 1]
```

4.2.7.11 `sorted()`

This function is global function. It accepts an iterable object, and will return a sorted version of the iterable object. It will not modify the original iterable object. We can pass an additional parameter `reverse=True` to indicate we want reverse sorted version (sorting in descending order). Example:

```
nums = [1, 2, 3, 4, 5]
print(nums)
sorted_num = sorted(nums, reverse=True)
print(sorted_num)
```

Output:

```
[1, 2, 3, 4, 5]
[5, 4, 3, 2, 1]
```

4.2.7.12 `.reverse()`

This function will reverse the order of its calling list. example:

```
nums = [3, 2, 7, 1, 5]
nums.reverse()
print(nums)
```

Output:

```
[5, 1, 7, 2, 3]
```

4.2.7.13 .copy()

This function will return a list, which is the copy of the calling list. Example:

```
nums = [3, 2, 7, 1, 5]
nums2 = nums.copy()
nums.reverse()
print(nums2)
```

Output:

```
[3, 2, 7, 1, 5]
```

More aboud Copying

Pay attention to how variable works in python. You can't use `nums2 = nums` to copy `nums` to `nums2`, since `nums2` will reference to the same memory space that actually store the content of the list (which is also referenced by `nums`).

Another way of copying a list is to use slice:

```
nums = [i for i in range(1, 10)]
nums2 = nums[:]
nums.reverse()
print(nums2)
```

Output:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

By not providing begin and end index, the slice will cover the whole list, it will return a **COPY** of the list (another chunk of memory will hold the copied content). Example:

```
nums = [i for i in range(1, 10)]
nums2 = nums[:]
nums.reverse()
```

```
print(nums2)
```

Output:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

4.2.7.14 `len()`

This is a global function. It accepts a list object, and will return the length of the list object (the number of items stored in the list). For example:

```
array = [1, 2, 3, 4, 5]
print(len(array))
```

Output:

```
5
```

`len()` can accept iterables. Like string, tuple, list, dictionary.

4.2.7.15 Number statistics for number list

A few python functions are specific to lists of numbers. For example: `min()`, `max()`, `sum()`. They are global functions. They accept list of numbers, and will return numeric type.

4.2.8 Tuples

Tuple is similar with list, but the element in tuple cannot be changed or modified, immutable. You can use tuple as immutable list (constant list).

4.2.8.1 Declaration

To declare a tuple in Python, you can use following syntax:

```
t1 = (1, 2, 3, 4, 5)
t2 = 5, "abc", 7, "today"
t3 = t2
print(t1)
```

```
print(t2)
print(t3)
```

Output:

```
(1, 2, 3, 4, 5)
(5, 'abc', 7, 'today')
(5, 'abc', 7, 'today')
```

4.3 Set

Set is an unordered collection of unique objects (no duplicate). An immutable version of set is frozenset.

4.3.1 Declaration

Use curly brackets to declare a set. For example:

```
my_set = {1, 3, 5, 7, 9, 9}
print(my_set)
```

Pay attention that I deliberately input a duplicate 9 in the curly bracket. Since set doesn't allow duplicate, there will be only one 9 inside `my_set`. Output:

```
{1, 3, 5, 7, 9}
```

You can also use the constructor of set to build up a set, just need to pass an iterable sequence into it. For example:

```
my_set2 = set(range(5, 15, 3))
print(my_set2)
```

Output:

```
{8, 11, 5, 14}
```

Notice that the output order is not exactly the same order we store the set. Python doesn't

care about the order in which each items in set is stored, it tracks only the connections between individual items. You can use `sorted()` method to sort the items in certain manner.

4.3.2 Functions

4.3.2.1 `sorted()`

This function is a global function. It accepts an iterable sequence, and will return a sorted version of the sequence. You can use it on set. For example:

```
set1 = {5, 2, 9, 8}
print(sorted(set1))
```

Output:

```
[2, 5, 8, 9]
```

4.4 Dictionary

The data structure Dictionary is similar with `std::map` in C++ STL. It can store key-value pair in a set, where each pair is located by its key. One difference between a python dictionary and a C++ map is, after the declaration of map, the data type of the key value pair of map is fixed. You can only add the same key-value pair into a specific map. While the dictionary in python allows you add different types of key-value pair

4.4.1 Declaration

To declare a dictionary, you are actually declaring a set of key-value pairs. You use the following syntax:

```
Dictionary = {
    key_1: value_1,
    key_2: value_2,
    ...
}
```

First enter the key, then a colon, then followed by the value corresponding to the key. Pay attention to the comma after each line of key-value pair. For example, declaring the following dictionary:

```
dictionary = {
    1: "ONE",
    "a": 65,
}
```

The keys in a dictionary should be unique.

Like `std::map` in C++, we can access the pair by providing the key directly in `operator[]`. Assume the above dictionary is defined:

```
print(dictionary[1])
print(dictionary["a"])
```

Output:

```
ONE
65
```

If you provide a nonexist key in `operator[]` without referencing it to a value, then an error will be thrown. To avoid error thrown, you can use member function `.get()` (introduced in functions section).

You can modify the value in the dictionary by using `operator[]` and the corresponding key value. For example:

```
dic1 = {
    "a": "First element",
    "b": "Second element",
    "c": "What is this?"
}

dic1["c"] = 3
```

```
print(dic1["c"])
```

Output:

```
3
```

4.4.2 Add New Key-value pair

To add new key-value pair into the dictionary, you would give the name of the dictionary followed by the new key in square brackets along with the new value:

```
dic1 = {
    "a": "First element",
    "b": "Second element",
    "c": "What is this?"
}

dic1["d"] = "Fourth element"
print(dic1)
```

Output:

```
{'a': 'First element', 'b': 'Second element',
'c': 'What is this?', 'd': 'Fourth element'}
```

4.4.3 Loop Through a Dictionary

4.4.3.1 Looping through key-value pairs

A dictionary can be traversed using its key or value, by calling the `.items()` member function. This function will return a "Dictionary items" object which is iterable, and each iterated item is a tuple containing `key` and `value`. You can use a for loop with two "iterating variables" to hold these two values and use them in the body of the for loop. Example:

```
dic1 = {
    "a": "First element",
```

```
    "b": "Second element",
    "c": "What is this?",
}

for key, value in dic1.items():
    print("Key: " + key + "; Value: " + value)
```

Output:

```
Key: a; Value: First element
Key: b; Value: Second element
Key: c; Value: What is this?
```

4.4.3.2 Looping through keys

To loop through keys, we use another function to return an iterable object which contains all the keys in the dictionary: `keys()`. The idea is the same as the key-value pair. Example:

```
dic1 = {
    "a": "First element",
    "b": "Second element",
    "c": "What is this?",
}

for key in dic1.keys():
    print(key)
```

Output:

```
a
b
c
```

Looping through the keys is actually the default behavior when looping through a dictionary, so this code would have exactly the same output if you wrote:

```
for key in dic1:
    print(key)
```

4.4.3.3 Looping through values

This is similar with looping through keys, except you use another member function `.values()` to get the iterable object which contains the list of each values in the dictionary's record. For example:

```
dic1 = {
    "b": "First element",
    "a": "Second element",
    "c": "What is this?",
}

for value in dic1.values():
    print(value)
```

Output:

```
First element
Second element
What is this?
```

4.4.4 Functions

4.4.4.1 `.get()`

This function accepts a key in the dictionary, it will then return the copied value associated with that provided key. If no key found, it will not throw an error, but return a `None`. For example:

```
dictionary = {
    1: "ONE",
    "a": 65,
```

```
4 }
5
6 print(dictionary.get(1))
7 print(dictionary.get("b"))
```

Output:

```
ONE
None
```

You can also determine the return value of this `.get()` function if the provided key is not found in the dictionary (other than None) (like the defaul value). Just put the return value in the second parameter slot when calling member function `.get()`. The default returned data can be anything legal.

```
1 dictionary = {
2     1: "ONE",
3     "a": 65,
4 }
5
6 print(dictionary.get(2, "No results found"))
7 print(dictionary.get("a"))
8 print(dictionary.get("c", ['n', 'o', ' ', 'r', 'e', 's', 'u', 'l', 't']))
```

The default return value in line 6 is a string, while in line 8 is a list. Output:

```
No results found
65
['n', 'o', ' ', 'r', 'e', 's', 'u', 'l', 't']
```

4.4.4.2 `len()`

This is a global function. It accepts an iterable object, and will return the length of the iterable object (the number of items stored in the object). When passing in a dictionary object, the number of key-value pairs will be returned. For example:

```
d = {
    1: 'a',
    2: 'b',
    3: 'c'
}
print(len(d))
```

Output:

```
3
```

4.4.4.3 `del`

`del` is not a function, it is a statement that can delete a key-value pair in dictionary. The syntax is as follows:

```
del dic1["key1"]
```

After the above statement, the key-value pair with key as `key1` is deleted.

4.4.4.4 `.items()`

This function will return a "dictionary items" type object, which contains all key-value pair in it. It is iterable, each iterated item is a **TUPLE** containing `key` and `value` (they cannot be changed). You can see this using following example:

```
dic1 = {
    "a": "First element",
    "b": "Second element",
    "c": "What is this?",
}

items = list(dic1.items())
print(items)
```

Output:

```
[('a', 'First element'), ('b', 'Second element'), ('c', 'What is this?')]
```

In the above code, the iterable "dictionary items" object has been used to construct a list, and referenced to `items`.

The iterable "dictionary items" object can be used in for loop to traverse the key-value pair in a dictionary.

4.4.4.5 .keys()

This function will return a `dict_keys` type object constructed by a list of keys.

```
dic1 = {
    "a": "First element",
    "b": "Second element",
    "c": "What is this?",
}

print(dic1.keys())
```

Output:

```
dict_keys(['a', 'b', 'c'])
```

4.4.4.6 .values()

This function will return a `dict_values` type object constructed by a list of values.

```
dic1 = {
    "b": "First element",
    "a": "Second element",
    "c": "What is this?",
}

print(dic1.values())
```

Output:

```
dict_values(['First element', 'Second element', 'What is this?'])
```

4.4.4.7 `.pop()`

This function accepts two parameters: the `key` and a default `value` to return if the key is not found in the dictionary. If you only provide the `key` parameter, you can still run the code, but in the case that the `key` is not in the dictionary, a `KeyError` will be thrown.

Chapter 5

Logic Control

5.1 `if` Statement

5.1.1 Syntax

The syntax for `if` statement is as follows:

```
raining = True

if raining:
    print("It's raining")
else:
    print("It's not raining")
```

Basically, you put the expression after the key word `if`, followed by a colon. The indentation rule is the same: multiple of four. Keyword `else` is used in the same way. The C++ `else if` contemporary in python is `elif`. For example:

```
if A:
    # do something
elif B:
    # do something
```

```
else:
    # do something
```

5.1.2 Key Word: `and`, `or`, `not`

Unlike C++ using `operator&&` and `operator||`, python just use keyword `or` and `and` to perform the same function. For example:

```
if expression_1 and expression_2:
    print("1 and 2 are both True!")
elif expression_1:
    print("1 is True and 2 is False")
elif expression_2:
    print("1 is False and 2 is True")
else:
    print("1 and 2 are both False")
```

The equivalent of C++'s not operator (`!`) in python is keyword: `not()`, for example:

```
expression = False
if not expression:
    print("The expression is False")
else:
    print("The expression is True")
```

Output:

```
The expression is False
```

5.1.3 Comparison Operators

The comparison operator is the same as C++:

- eual: ==
- greater: >

- smaller: `<`
- greater than or equal to: `>=`
- smaller than or equal to: `<=`
- not equal: `!=`

Example:

```
def max_num(num1, num2, num3):
    if num1 >= num2 and num1 >= num3:
        return num1
    elif num2 >= num3:
        return num2
    else:
        return num3
```

5.1.4 Check if an object is in a container

We have a handy way to check if an object is in a container, by using keyword `in`.‘ For example:

```
array = [1, 2, 3, 4, 5]
item = 'a'
if item in array:
    print(item, "is in the array!")
else:
    print(item, "is not in the array!")
```

Output:

```
a is not in the array!
```

Similarly, you can use `not in` to check if an object is not in a container:

```
if item not in array:
```

```
    # do something here
```

Another example:

```
nums = list([2, 3, 4, 5, 13, 6, 8])
guess_nums = list([5, 7])

for guess_num in guess_nums:
    if guess_num in nums:
        print("you got a lucky number: " + str(guess_num))
    else:
        print("Sorry, but", guess_num, "is not a lucky number")
```

Output:

```
you got a lucky number: 5
Sorry, but 7 is not a lucky number
```

5.1.5 Check if a container is empty

You can use if statement to check if a container is empty. An empty container will be treated as `False`, for example:

```
string = ""
nums = list()

if string:
    print(string)
else:
    print("The string is empty!")

if nums:
    print(nums)
else:
```

```
12      print("The list is empty!")
```

Output:

```
The string is empty!
The list is empty!
```

5.2 while Loop

5.2.1 Syntax

The syntax of while loop in python is:

```
1  while expression:
2      # do something here
3      # do something here
```

Don't forget the colon after the expression!

Example: print integer from i = 1 to 10:

```
1  i = 1
2  while i <= 10:
3      print(i)
4      i += 1
```

Similar with C++, python uses `break` as the keyword to stop the loop, `continue` as the keyword to stop current iteration and begin next iteration.

5.2.2 while and list

The logic expression after the `while` keyword can be a mutable container, if the size of the container is zero, the while loop will stop. This is true for the list container. For example:

```
1  nums = [1, 2, 3]
2  while nums:
```

```
3     print(nums.pop())
```

Output:

```
3
2
1
```

5.3 `for` loop

The `for` loop in python works like the range based for loop in C++.

5.3.1 Syntax

The `for` loop's syntax is:

```
1 for letter in "abcde":
2     print(letter)
```

Don't forget the colon at the end of the line.

Basically, the object after the `in` keyword should be iterable. And the variable right after `for` keyword will go through the iterable and copy each object in the iterable object to the variable (just like C++'s range based for loop, not referenced). Output of the above loop is:

```
a
b
c
d
e
```

You can also specify the range of the for loop using integer value, by calling the `range()` function. For example:

```
1 for i in range(5):o
2     print(i)
```

Output:

```
0
1
2
3
4
```

`i` will go through 0 to $N-1$.

To specify the range, you use comma, for example:

```
for i in range(5, 10):
    print(i)
```

Output:

```
5
6
7
8
9
```

Not including the last limit.

Another example:

```
d = [1, 2, 'a', 5.5]
for index in range(len(d)):
    print(d[index])
```

Output:

```
1
2
a
5.5
```

5.3.2 More About `range()`

`range()` method will return an iterable `range` type object. You can convert it into a list by passing this object to `list()`. `list()` is type constructor that can accept an iterable object, and return a list that is constructed by each iterated item in the iterable object.

Example:

```
chars = list("abcde")
nums = list(range(1, 10, 2))
print(chars)
print(nums)
```

Output:

```
['a', 'b', 'c', 'd', 'e']
[1, 3, 5, 7, 9]
```

5.3.3 Traversing a Multi-dimension Iterable Object

As discussed in Syntax section, we can use a for loop to easily traverse an iterable object:

```
nums = [1, 2, 3, 4]
for num in nums:
    print(num)
```

We may encounter situations that each iterated item is itself an iterable, for example:

```
coordinates = [(1, 2), (3, 4), (5, 6)]

for x in coordinates:
    print(x)
```

In the above code, each iterated item is actually a tuple. The output is as follows:

```
(1, 2)
(3, 4)
```

```
(5, 6)
```

Each tuple is printed as a tuple. However, its possible that use two variables to **UNPACK** the tuple during the for loop, so we can work with each value in the tuple individually, the syntax is as follows:

```
coordinates = [(1, 2), (3, 4), (5, 6)]

for x, y in coordinates:
    print("Point", coordinates.index((x, y)) + 1, " x:", x, "y:", y)
```

`x` and `y` will reference to the corresponding element in the Tuple itself! Output:

```
Point 1  x: 1 y: 2
Point 2  x: 3 y: 4
Point 3  x: 5 y: 6
```

However, the number of variables used to **unpack** should be **equal** to the number of elements in each iterated object. In the above code, each tuple has two elements, so we use two variables to unpack it. If they are not equal, there will be three possible scenario.

1

Number of variable is 1, number of elements in iterated item is more than one. In this case, the object can't be unpacked. The variable will reference the object as a whole (for example, the tuple). This is shown in the leading example.

2

Number of variable is not 1, but less than the number of elements in iterated item. For example:

```
coordinates = [(1, 2, 3), (3, 4, 4), (5, 6, 9)]

for x, y in coordinates:
    print("x:", x, "y:", y)
```

In this case, a `ValueError` will be thrown:

```
ValueError: too many values to unpack (expected 2)
```

Python expects to unpack two values (because there is only two variables to be used to hold the unpacked value), but it encounters three.

3

Number of variable is more than the number of elements in iterated item. For example:

```
coordinates = [(1, 2, 3), (3, 4, 4), (5, 6, 9)]

for x, y, z, m in coordinates:
    print("x:", x, "y:", y)
```

In this case, a `ValueError` will be thrown:

```
ValueError: not enough values to unpack (expected 4, got 3)
```

Python expects to unpack four values (because there are four variables to be used to hold the unpacked value), but it encounters three.

Chapter 6

Functions

6.1 Declaration

To declare a function, use keyword `def`, followed by the function name and a `:`.

```
1 def hello():
```

The parenthese will keep the parameter of the function, followed by the `:`. Unlike C++, python use indentation to delimit different code block. For example, we can define the following function:

```
1 def hello():
2     print("Hello! please input a line: ")
3     string = input()
4     print("You have inputed: ", string)
```

Pay attention that python uses indentation as a way to define scope. PEP8 requires that indentation is a multiple of 4.

Other rules:

- Function name should be all lower case, connected by under score symbol.
- There should be a two empty line between the end of definition of a function and the

next code

To declare a function with parameters, you don't have to declare the type of the variable, you directly write the name of the variable you are going to use inside the function definition:

```
1 def say_hi(name):
2     print("Hello ", name)
3
4
5 say_hi(input("Please input your name: "))
```

Output:

```
Please input your name: Miao
Hello  Miao
```

The function can return a value, just use `return` keyword, similar with C++. You don't have to declare the **RETURN TYPE** of the function.

6.2 Passing Arguments

You can pass arguments to functions in a number of ways:

- positional arguments: arguments passed in should be the same order the parameters were written
- keyword arguments: each argument consists of a variable name and a value
- arguments are lists and dictionaries of values

6.2.1 Positional Arguments

This is the original way of passing arguments to a function. You provide the variable in the function call's parenthesis. For example, if you defined a function as follows:

```
1 def my_function(A, B, C):
2     """sample function"""
```

and then, you cann the function by:

```
1 my_function(a, b, c)
```

You just provided three variables in the parenthese. Python will interpret these as the positional arguments, and will relate `a` to the first parameter: `A`, relate `b` to the second parameter: `B`, relate `c` to the third parameter: `C`. Order of the arguments is crucial.

6.2.2 Keyword Arguments

A keyword argument is a name-value pair that you pass to a function, i.e. you specify which parameter takes which argument. You have to know the exact name of parameter when defining the function, and use an assignment operator to pass the corresponding argument when calling the function. Example:

```
1 def print_name(name, age):
2     print(name + " is", age, "years old")
3
4 print_name(name = "Yu", age = 29)
5 print_name(age = 29, name = "Yu")
```

Output:

```
Yu is 29 years old
Yu is 29 years old
```

Keyword argument doesn't require arguments passed in the same order defined in the function, so the output of line 4 and line 5 are the same.

6.2.3 Default Arguments for Parameters

Similar with C++, you can define default value for a function's parameter. Syntax:

```
1 def print_name(age, name="friend"):
2     print(name + " is", age, "years old")
3
```

```

print_name(23)
```

Output:

```
friend is 23 years old
```

Parameters with default argument should be placed after parameters without default parameter. This allows python to continue interpreting positional arguments correctly. Also, don't include spaces on left and right of the parenthese when defining default value (PEP8).

6.2.4 Passing an Arbitrary Number of Arguments

You can define a function that takes an arbitrary number of arguments. To do this, you use following syntax:

```
def print_input(*string):
    print(string)
```

The asterisk in the parameter name `*string` tells python to make an empty tuple called `string` and pack whatever values it receives into this tuple. You can think of the function receives a tuple filled with arguments, you can work around the tuple to get your desired output. The `print()` function is an example that accepts an arbitrary number of arguments. If you are using an IDE, when you input the arguments for `print()` function, you'll notice the prompt of the function parameter has an asterisk, which means you can provide an Arbitrary number of parameters to it.

6.2.5 Positional & Arbitrary Number of Arguments

It is possible to write a function that accepts positional and arbiturary arguments at the same time. You just need to place the parameter that accepts arbitrary number of arguments in the last of the function parameter list. Python matches positional and keyword arguments first and then collects any remaining arguments in the final parameter. Example:

```
def print_lucky_number(name, age, *lucky_number):
    print("The lucky number for " + name + " (" + age + " years old)" + "
    ↪  is:\n")
    for num in lucky_number:
        print(num)


print_lucky_number("Yu", "29", 5, 3, 1, 9, 8, 7, 55)
```

Output:

```
The lucky number for Yu (29 years old) is:


5
3
1
9
8
7
55
```

6.2.6 Arbitrary Key-Value Pair Arguments

Sometimes you'll want to accept an arbitrary number of arguments, but you won't know ahead of time what kind of information will be passed to the function. In this case, you can write functions that accept as many key-value pairs as the calling statement provides. The syntax is as follows:

```
def build_profile(first, last, **user_info):
    """build a dictionary containing everything we know about a user"""
    profile = {}
    profile['first_name'] = first
```

```
    profile['last_name'] = last
    # push in the arbitrary key-value pairs into dictionary
    for key, value in user_info.items():
        profile[key] = value
    return profile


user_profile = build_profile("Yu", "Miao",
                             location='Tallahassee',
                             field="Computer Science",
                             age=29)
print(user_profile)
```

Output:

```
{'first_name': 'Yu', 'last_name': 'Miao',
'location': 'Tallahassee',
'field': 'Computer Science', 'age': 29}
```

The double asterisks before the parameter `**user_info` cause python to create an empty dictionary called `user_info` and pack whatever name-value pairs it receives into this dictionary. Within the function, you can access the name-value pairs in `user_info` just as you would for any dictionary. In the above example, you can think of the following version of `build_profile()` function is actually declared in runtime (when calling):

```
def build_profile(first, last, location, field, age):
    # function body goes here
```

Another simpler example:

```
def print_info(**info_dict):
    for key, value in info_dict.items():
        print(key, ":", value)

```

```
5
6 print_info(name="Yu Miao", age=29, field="Computer Science")
```

Output:

```
name : Yu Miao
age : 29
field : Computer Science
```

This function is actually doing the same thing as:

```
1  def print_info(name, age, field):
2      info_dict = {}
3      info_dict['name'] = name
4      info_dict['age'] = int(age)
5      info_dict['field'] = field
6
7      for key, value in info_dict.items():
8          print(key, ":", value)
9
10
11 print_info(name="Yu Miao", age=29, field="Computer Science")
```

Output:

```
name : Yu Miao
age : 29
field : Computer Science
```

So you may not think these key-value pairs as parameter-argument pairs and use them as the normal way of parameter-argument pairs do. Because the function doesn't know the type or value that are going to be passed into the function, so there is little the function can do with those passed in key-value pairs except store them as data.

6.2.7 Default and Arbitrary Number of Arguments

May only work with python3. Reference for this section can be found here.

You can mark a parameter with default, and the other parameter as arbitrary number at the same time in a function definition. Example:

```
def print_info(*favorite_languages, name="Friend"):
    print(name + "'s favorite languages are: ")
    for language in favorite_languages:
        print(language.title())


print_info("c++", "python", "c", "matlab")
```

Output:

```
Friend's favorite languages are:
C++
Python
C
Matlab
```

Notice that you have to put the arbitrary parameter before the default parameter. When you call the function, you pass the arbitrary parameter directly. If you want to pass another value to the default parameter, you have to use the keyword arguments syntax:

```
print_info("c++", "python", "c", "matlab", name="Yu")
```

Output:

```
Yu's favorite languages are:
C++
Python
C
Matlab
```

6.3 Styling Functions

There are several styling rules when writting a python function:

6.3.1 Use descriptive names

Function names should be clear of what it does.

6.3.2 function_name

Function names should use lowercase letters and underscores only (module names should use these conventions as well). PEP8 recommends that you limit lines of code to 79 characters so every line is visible in a reasonably sized editor window. If a set of parameters causes a function's defnition to be longer than 79 characters, press enter after the opening parenthesis on the defnition line. On the next line, press tab twice to separate the list of arguments from the body of the function, which will only be indented one level.

6.3.3 docstring

Every function should have a docstring immediately after the function definition, which explains concisely what the function does. Others should be able to use your function just by looking at your docstring. They should be able to trust that the code works as described, and as long as they know the name of the function, the arguments it needs, and the kind of value it returns, they should be able to use it in their programs.

6.3.4 spaces between functions

If your program or module has more than one function, you can separate each by two blank lines to make it easier to see where one function ends and the next one begins.

Chapter 7

Input and Output

7.1 Input

To get an input from keyboard, you can just use:

```
input("Prompt information goes here")
```

The string of this function is the message that will show to prompt user to type in the information. This function will return a **STRING** object which containing what you just inputted from the keyboard. You can declare a variable to hold the content that has been typed in, for example:

```
1 num = input("Please input a number:")
2 print("You inputted: " + num)
```

Output:

```
Please input a number: 5
You inputted: 5
```

Unlike C++'s `std::cin »`, which is actually an `operator»()` that accepts a `std::cin` object, `input()` in python is a function that returns the string you inputted. This makes the following operation possible:

```
print("You have inputted: " +
  input("Please input first number: ") +
  " and " +
  input("Please input second number: "))
```

In the above code, the first part `"You have inputted:  "` will not be printed out first. All items will be printed only after all to-be-printed items are determined. The sequence of evaluating un-determined items are from left to right. So, for the above code, the output will be:

```
Please input first number: 2
Please input second number: 3
You have inputted: 2 and 3
```

It seems that python somehow **STORED** the returned string from the `input()` function somewhere, so they can be used when `print()` actually execute. This is not like C++, because in C++ if you don't use a container to hold the returned value from a function, it will gone (like rvalue?).

Chapter 8

Catching Errors and Exception

8.1 General Idea

When an error is thrown, program will stop running. But we have strategy to deal with that. The idea is same as catching exceptions in C++.

We use try and except block to handle this situation. The most basic syntax is as follows:

```
try:
    number = int(input("Enter a number: "))
    print(number)
except:
    print("Invalid input")
```

If any error was thrown in the try block, the control will go into the except block, code in except block will be executed.

When writing python code, we should catch the specific types of error that has been thrown. All we have to do is to specify the kinds of error in the except line. For example:

```
try:
    number = int(input("Enter a number to do a division: "))
    # if number is zero, ZeroDivisionError will be thrown
```

```
    print("The result of 2 /", number, "is", 2 / number)

    # if non-integer value is entered, ValueError will be thrown
    number = int(input("Enter an integer to display: "))
    print(number)

except ZeroDivisionError:
    print("Can't divide by zero!")
except ValueError:
    print("Non-integer value inputted!")
```

However, once an error is thrown in the try block, the remaining section of the try block will not be executed.

We can store the error thrown by the try block using a variable, and print out the variable directly to show what the error is. Keyword `as` is used, for example:

```
try:
    number = int(input("Enter a number to do a division: "))
    # if number is zero, ZeroDivisionError will be thrown
    print("The result of 2 /", number, "is", 2 / number)

    # if non-integer value is entered, ValueError will be thrown
    number = int(input("Enter an integer to display: "))
    print(number)

except ZeroDivisionError as err:
    print(err)
except ValueError as err:
    print(err)
```

8.2 The `else` Block

We can use following structure to write our try-except block:

```
num1 = float(input("Enter first number: "))
num2 = float(input("Enter second number: "))

try:
    answer = num1 / num2
except ZeroDivisionError:
    print("Can't divide by 0!")
else:
    print(num1, "/", num2, " = ", answer)
```

We have used `else` to store all the operations we want to do if the `try` block didn't trigger any error. The only code that should go in a try statement is code that might cause an exception to be raised.

8.3 Failing Silently

You can add `pass` in the except block, so python will do nothing once an error is thrown.

8.4 Python Errors

8.4.1 `NameError`

A name error usually means we either forgot to set a variable's value before using it, or we made a spelling mistake when entering the variable's name.

8.4.2 SyntaxError

8.4.3 TypeError

8.4.4 IndexError

8.4.5 IndentationError

8.4.6 ZeroDivisionError

8.4.7 FileNotFoundError

Chapter 9

File Operations

9.1 Open a File

9.1.1 Manually Close

To work with file, you have to first open a file. Use following syntax to open a file:

```
1 open("time.txt", "r")
```

`open()` is a global function, the first parameter is the path of the file to be opened. It can be relative path, exact path or just the file name. The second parameter is the mode you open the file with (similar with C++ file object). Differen meanings are as follows:

parameter	meaning	notes
"r"	read only	read only, old info kept
"w"	write only	write only, old info cleared
"a"	write only	writing starts from the end of the old file
"r+"	read & write	old info kept
"w+"	write & read	old info cleared
"a+"	write & read	writing starts from the end of the old file

The default mode is read mode.

You can use a variable to hold the file you opened:

```
time_file = open("time.txt", "r")
time_file.close()
```

(or does it act like a file object? like in C++? The `open()` function will return a constructed file object, and you can "catch" it using a variable) In the above code, you close the file by calling the `close()` member function of `time_file`.

9.1.2 Automatically Close

We can use following syntax to open a file, and do things within the same block. We don't have to manually close the file. When the control flow goes out of the block, the file is closed:

```
with open("pi_digits.txt") as file_object:
    content = file_object.readline()
    print(content)
```

Output:

```
3.1415926535
```

9.1.3 File Path

When open a file, you have to provide the file path. There are three possible ways to provide a file path:

1. by providing the name of the file; this requires the file is at the same folder of the program
2. by providing the *relative path*
3. by providing the *absolute path*

9.1.3.1 File name

If the file is in the same folder of the program source code, you can just provide the file name. For example:

```
file_object = open("pi_digits.txt")
```

9.1.3.2 Relative path

If the file is stored in sub-folder that are inside the program's folder, you can provide the *relative path* of where the file is (pinpoint to the file name). The format of file path is different on Windows and OSX or Linux.

Example (Windows):

```
file_object = open("text_files\file_name.txt")
```

Example (OSX or Linux):

```
file_object = open("text_files/file_name.txt")
```

9.1.3.3 Absolute path

Example (Windows):

```
file_object = open("C:\Users\My\other_files\text_files\filename.txt")
```

Example (OSX or Linux):

```
file_object = open("/home/My/other_files/text_files/filename.txt")
```

9.1.4 Check if File Existed or Not

It is quite often that a file is not found when open a file. It is a good practice to use try-except blocks to catch this kind of error. Example:

```
file_opened = False

while not file_opened:
    file_path = input("Please input file path to open: ")
    try:
        file_object = open(file_path)
```

```
7       except FileNotFoundError:
8           print("Invalid path, file open failed")
9       else:
10          file_opened = True
11          print(file_object.read())
12          file_object.close()
```

9.2 Reading Line by Line

Aside from using member functions to read the file line by line, you can simply use a for loop to do the job. For example:

```
1  file_object = open("pi_digits.txt")
2
3  for line in file_object:
4      print(line)
5
6  file_object.close()
```

Output:

```
3.1415926535

8979323846

2643383279
```

These blank lines appear because there is a `'\n'` at the end of each line in the text file. The `print()` function adds its own newline each time we call it, so we end up with two newline characters. We can use `.strip()` method to eliminate extra whitespace in the string.

9.3 File object functions

9.3.1 Read

When you need to read a file, open it in mode that supports reading (`"r"`, `"r+"`, `"a+"`, `"w+"`). Following functions can be used to read from the file in different ways. Pay attention to the position of where the file object at.

9.3.1.1 .readable()

This function will return a Boolean value, which indicates whether the file is readable or not. For example, if a file has been opened in `"w"` mode, it cannot be read, so following result will be `False`:

```
time_file = open("time.txt", "w")
print(time_file.readable())
```

9.3.1.2 .read()

This function will return the copy of the file (in the form of a long string, including special characters like `'\n'`). You can store it in a variable, for example:

```
time_file = open("time.txt", "r")
file_content = time_file.read()
print(file_content)
```

The above code will print the whole content of `time.txt`.

This function requires the file to be readable (open in right mode).

9.3.1.3 .readline()

This function will read the file one line at a time (read the string until a `'\n'` is met, the newline character is read. Unlike `getline()` function in C++). It will update the position of the file object. For example:

```
time_file = open("time.txt", "r")

file_content = time_file.readline()
print(file_content)

file_content = time_file.readline()
print(file_content)

file_content = time_file.readline()
print(file_content)

time_file.close()
```

The first three lines of time.txt will be printed out.

This function requires the file to be readable (open in right mode).

9.3.1.4 .readlines()

This function will read the file line by line, and construct a list whose elements are each line in the file. When finish reading, it will return this list. For example:

```
time_file = open("time.txt", "r")

file_content = time_file.readlines()
print(file_content)

time_file.close()
```

Output:

```
['abcde\n', '12345\n']
```

Pay attention that, the newline character is also read.

You can use a bracket operator and an integer to specify the line you want to read. For example:

```
time_file = open("time.txt", "r")
file_content = time_file.readlines()[1]
print(file_content)
```

Only the second line will be read. Also, after this line, the file object will reach the end of the file, and you can't use `read()`, `readline()`, `readlines()` unless you reset the file pointer to the beginning by calling `seek(0)`.

Also, the integer value in the bracket is the relative position of the line from the current file object position in the file. For example:

```
time_file = open("time.txt", "r")

file_content = time_file.readline()
print(file_content)

file_content = time_file.readlines()[0]
print(file_content)

time_file.close()
```

The code `time_file.readlines()[0]` will actually return the second line in the file. Because when it is called, the file pointer is already in the beginning of the second line.

This function requires the file to be readable (open in right mode).

9.3.2 Write

When you need to write to a file, open it in mode that supports writting (`"w"`, `"a"`, `"w+"`, `"a+"`). Pay attention to the position of where the file object at.

9.3.2.1 .write()

This function accepts a string object, it will write the string to the file. Python can only write strings to a text fle. If you want to store numerical data in a text fle, you'll have to convert the data to string format frst using the str() function.

This function does not add any newlines to the text you write. You have to add it by yourself.

9.3.3 Navigate

9.3.3.1 .seek()

This function accepts an integer indicating the position of the file object to navigate to. 0 means move to the first character, 1 means move to the second character, etc. For example, the time.txt is:

```
abcde
12345
```

We run the following code:

```
time_file = open("time.txt", "r")

file_content = time_file.readlines()[1]
print(file_content)

time_file.seek(0)

file_content = time_file.readline()
print(file_content)

time_file.close()
```

Output:

```
12345

abcde
```

`.seek(0)` is called to reset the file object position (which is at end of the file after the calling of `.readlines()[1]`).

9.4 Storing Data with `json` Module

The JSON (JavaScript Object Notation) format was originally developed for JavaScript. However, it has since become a common format used by many languages, including Python. The `json` module allows you to dump simple Python data structures into a file and load the data from that fle the next time the program runs.

There are two methods defined in `json` module:

- `json.dump()`: used to write data to a file
- `json.load()`: used to load data from a file (created by ~json.dump())

In order to use these methods, you have to import the `json` module:

```
1 import json
```

9.4.1 Write to File

We use `json.dump()` to write to file. This method accepts two arguments: the first one is the data structure you want to write to file, the second one is the file object which opened the file you wish to write to (in `"w"` mode). For example:

```
1 import json
2 nums = [2, 4, 6, 7, 8]
3 file_object = open("nums.json", "w")
4 json.dump(nums, file_object)
```

```
file_object.close()
```

The above code has created a list, and used `json.dump()` to write the content in the list to a file.

9.4.2 Load from File

We use `json.load()` to load from file. This method accepts one argument: the file object which opened the file to be read. It will return the data structure contained in the file, which can be "catched" by a variable. For example:

```
file_object = open("nums.json")
nums = json.load(file_object)
print(nums)
```

Output:

```
[2, 4, 6, 7, 8]
```

9.4.3 Saving and Reading User-Generated Data

We can save our data to disk and retrieve later to do more things. The following example shows a simple application. We can ask a user to input the username, and send greeting information based on the fact that whether or not the name has been encountered before.

The textbook also talks about the idea of *refactoring*, which means divide your program into smaller components, each component should have one clear, concise functionality, described by a clear self-descriptive name. Then the whole program should be clear.

Example:

```
import json


def load_user_name():
    """
```

```
6       This function tries to load saved user info (in user_info.json)
7       1. if open successfully, return the stored list using json.load()
8       2. if not, which means no previous record found, return an empty list
9       """
10      try:
11          file_object = open("user_info.json")
12      except FileNotFoundError:
13          return []
14      else:
15          return json.load(file_object)
16
17
18  def store_user_name(user_names):
19      """
20      store updated user_names to user_info.json
21      """
22      file_object = open("user_info.json", "w")
23      json.dump(user_names, file_object)
24
25
26  def greet_user():
27      """
28      This function will first ask for a user name, then send greeting
    ↪   information
29      based on whether or not user name has encountered before
30      :return: None
31      """
32      user_name = input("Input username: ")
33      user_names = load_user_name()
34      if user_name in user_names:
```

```
35          print("Welcome back,", user_name)
36      else:
37          user_names.append(user_name)
38          store_user_name(user_names)
39          print("We'll remember you when you come back,", user_name)
40
41
42  greet_user()
```

Chapter 10

Class

10.1 Declaration

To declare a class, you use following syntax:

```
1 class Student:
2
3     def __init__(self, name, major, gpa, is_on_probation):
4         self.name = name
5         self.major = major
6         self.gpa = gpa
7         self.is_on_probation = is_on_probation
```

The `__init__` is a special method in python. It is like constructor in C++, which is called when we create an instance of the class.

10.2 Defining Data Member in Class

Unlike C++, we directly define the data member of the class in the body of `__init__()`. `__init__()` can accept arguments passed in and use them to declare a data member (in python, its called class's attribute), or it can declare attributes inside its body, just providing

some initial values. For example:

```
def __init__(self, make, model, year):
    """initialize attributes to describe a car"""
    self.make = make
    self.model = model
    self.year = year
    self.odometer = 0
```

In the above example, the attribute `self.odometer` is defined inside the body of `__init()__`. You can also:

```
def __init__(self, make, model, year, odometer=0):
    """initialize attributes to describe a car"""
    self.make = make
    self.model = model
    self.year = year
    self.odometer = odometer
```

One thing should be made clear: every attribute in a class needs an initial value, even if that value is 0 or an empty string, or other empty stuff.

10.3 Constructing an Instance

After we declared a class, we use following syntax to construct an instance of the class:

```
student1 = Student("Yu", "Computer Science", 4.0, False)
```

We are calling the `__init__()` and passing in necessary parameters to build our class object.

10.4 Inheritance

10.4.1 Syntax

Assume we have a base class as follows:

```
class Car:
    """A simple attempt to represent a car"""

    def __init__(self, make, model, year, odometer=0):
        """initialize attributes to describe a car"""
        self.make = make
        self.model = model
        self.year = year
        self.odometer = odometer
        # self.odometer = 0

    def get_descriptive_name(self):
        """return a formatted descriptive name"""
        long_name = str(self.year) + ' ' + self.make + ' ' + self.model
        return long_name.title()

    def read_odometer(self):
        """print a statement showing the car's odometer"""
        print("This car has " + str(self.odometer) + " miles on it")
```

Now, we want to define another class `ElectricCar`. This class can be defined as Inheritance of the base `Car` class (as its child class). The first line to define it would be:

```
class ElectricCar(Car):
```

Pay attention we put the parent class `Car` into a parenthese followed by the name of child class. You have to make sure the parent class's definition is inside the file where you define `ElectricCar` class, and should be before the definition of `ElectricCar` class.

By default, the parent class's `__init__()` is called and attributes constructed. You don't have to write another `__init__()` function. However, if you want to define new attributes, you have to write the child class's `__init__()`. Inside which, you have to explicitly call

parent class's `__init__()`. Assume we want to add one more attribute to the `ElectricCar` class: `battery_level`, which has a default value of 100, then we can write child class's `__init__()` as:

```
1 def __init__(self, make, model, year, odometer=0):
2     super().__init__(make, model, year, odometer)
3     self.battery_level = 100
```

Notice that on line 2, `super()`'s member function `__init__()` is called and arguments passed. `super()` is actually the parent class of this `ElectricCar` class, which is `Car` class. The name *super* comes from a convention of calling the parent class a *superclass* and the child class as a *subclass*.

10.4.2 Overriding Methods from the Parent Class

You can override any method from the parent class that doesn't fit what you're trying to model with the child class. Just redefine the method in the body of child class. Python will disregard the parent class method and only pay attention to the method you define in the child class.

10.5 Styling Classes

10.5.1 Class Names

Class names should be written in *CamelCaps*.

10.5.2 Docstring

Every class should have a docstring immediately following the class definition. Also, each module should have a docstring describing what the classes in that module can be used for.

10.5.3 Blank Lines inside Module

Insert one blank line between methods in the same class.

Insert two blank lines between classes in the same module.

Chapter 11

Manage External Code (Module)

A module is a python file we can import into our current python file.

11.1 `import`

You can import external code in your python program. For example, to use some math functions defined in python math, you can type:

```
1 from math import *
```

In this way you imported the `math` module into your program, and you can use the functions predefined in the module directly. For example:

```
1 from math import *
2 print(floor(2.3))
```

The `*` indicates you import the whole module from `math`. You can also import just one element in the module, by using:

```
1 from math import floor
```

Then, you only import the component named `floor`.

Another way to import the file: assume you have a file named `useful_file.py` (in your

current working path). To import it to your code, you just need to:

```
import useful_file
```

No need to write the extension. Using this method to import, when you have to access members in the file, you have to use dot operator (`.`). For example, assume in `useful_file`, a function named `roll_dice()` is defined, then you use the following syntax to use the function in your code:

```
import useful_file
print(useful_file.roll_dice(10))
```

Another thing should be noticed is that, when importing module, its like copy all the content in the module into where you import it (so after the import line, you can use things defined in the module). When you run the program, anything in the module will also be run through. For example, if your module is like:

```
import random
print("This is a module")
def roll_dice(num):
    return random.randint(1, num)
```

Then, if your code is:

```
import useful_file
print(useful_file.roll_dice(10))
```

Then you'll actually have the following output:

```
This is a module
5 <generated random number>
```

This is not the same as `#include ""` in C++, since any call of function happens in the `main()` function, which is unique in a C++ program. In python, you don't need a `main()` function to do stuff like `print()`.

To avoid namespaces corruption, it is better to import the function or functions you want,

or import the entire module and use the dor notation.

PEP8: All import statements should be written at the beginning of a fle. The only exception is if you use comments at the beginning of your fle to describe the overall program.

11.1.1 Using as

11.1.1.1 Give an Alias to a Function

If the name of a function you're importing might conflict with an existing name in your program or if the function name is long, you can use a short, unique *alias* for the function. You'll give the function this special nickname when you import the function. For example:

```
1 from useful_file import roll_dice as rd
```

Then, you can call `roll_dice()` function by `rd()`.

11.1.1.2 Give an Alias to a Module

You can also provide an alias for a module name. This allows you to call the module's functions more quickly. For example:

```
1 import useful_file as uf
2 uf.roll_dice()
```

Apparently, `uf.roll_dice()` is more concise than `useful_file.roll_dice()`.

11.2 `import` Classes

You use following syntax to import class in a module:

```
1 from module_name import class_name
```

If you want to import multiple classes, do:

```
1 from module_name import class_name_1, class_name_2
```

If you want to import all classes, use asterisk:

```
from module_name import *
```

However, this is not encouraged, beside the name corruption reason, there is another reason: it is beneficial to actual see which class or method you use in the imported class.

To avoid namespace corruption, you may also want to:

```
import module_name
# or:
import module_name as m_n
```

and then access the class by using dot operator.

11.3 Using `pip`

11.3.1 `pip` for python2 and python3

To install modules by pip for python3, use:

```
sudo python3 -m pip install <package name>
```

(The default python version on my ubuntu is python 2.7)

Or use:

```
pip3 install <package name>
```

To uninstall, just replace the keyword `install` with `uninstall`.

11.4 Install packages in pycharm

To install a package in pycharm, follow:

```
File -> Setting -> Project: -> Project interpreter -> +
-> search the packages -> install
```

(For some reason, I can't install by pip and use in pycharm. Maybe its related to path issue)

11.5 Styling

If you need to import a module from the standard library and a module that you wrote, import sequence is:

1. module from the standard library
2. a blank line
3. module that you wrote

Chapter 12

Code Test

12.1 Testing a Function

Many times you need to test the code using different test cases. The module `unittest` from the python standard library provides tools for testing your code. A `unit test` verifies that one specific aspect of a function's behavior is correct. A `test case` is composed of different `unit cases` that together prove that a function behaves as its supposed to.

12.1.1 Steps

Suppose you have a function to test, the function is defined in module `name_function`, its definition is as follows:

```
1 def get_formatted_name(first, last):
2     """Generate a neatly formatted full name"""
3     full_name = first + " " + last
4     return full_name.title()
```

In your test file (which contains code to run test cases for this function), you first import the `unittest` module and the module contains your function:

```
1 import unittest
```

```
2 from name_function import get_formatted_name
```

Then, you have to define a child class which is inherited from a class in `unittest` named `TestCase`:

```
1 class NamesTestCase(unittest.TestCase):
```

The name of the class is arbitrary, however it is a good practice to choose names that are self-explanatory: the name should be related to the function you are about to test and to use the word *Test*.

Inside the class, we define a single method that tests one aspect of `get_formatted_name()`:

```
1 def test_first_last_name(self):
2     """Check combining and title()"""
3     formatted_name = get_formatted_name('yu', 'miao')
4     self.assertEqual(formatted_name, 'Yu Miao')
```

The name of this method is to indicate that we are testing names that contains only first and last name (no middle name). Any method that starts with `test_` will be run automatically when we run the program that containing this test class (for example, the above method is starting with `test_` and the full name is `test_first_last_name`).

In this test method, at line 3, we call the function we want to test and pass in necessary arguments. Then we store the return value from the function. After that, at line 4, we use `assert` method to check check the returned value by the function. In this case, we call `assertEqual()` to check the if the returned value is equal to what we expect from the function giving the input at line 2. The first argument of `assertEqual()` is the returned value from the function, the second argument of `assertEqual()` is the expected value.

After the definition of the class, we have to use following line to actually run the test:

```
1 unittest.main()
```

The whole test file is shown below:

```
1 import unittest
```

```
2  from name_function import get_formatted_name
3
4
5  class NamesTestCase(unittest.TestCase):
6      """Test for 'name_function.py"""
7
8      def test_first_last_name(self):
9          """Check combining and title()"""
10         formatted_name = get_formatted_name('yu', 'miao')
11         self.assertEqual(formatted_name, 'Yu Miao')
12
13
14 unittest.main()
```

Output:

```
.
----------------------------------------
Ran 1 test in 0.000s

OK
```

The dot on the first line of the output tells us that a single test passed. The next line tells us that python ran one test, and the corresponding time used to test the case. The final `OK` tells us that all unit tests in the test case passed.

12.1.2 Failing Test

A test can fail for different reasons. Take the above as an example. The expected outcome is like "Yu Miao", we can change this to "YuMiao", so we know test will fail. i.e. we rewrite the test file as follows:

```python
import unittest
from name_function import get_formatted_name


class NamesTestCase(unittest.TestCase):
    """Test for 'name_function.py'"""

    def test_first_last_name(self):
        """Check combining and title()"""
        formatted_name = get_formatted_name('yu', 'miao')
        self.assertEqual(formatted_name, 'Yu Miao')


unittest.main()
```

Output:

```
F
======================================================================
FAIL: test_first_last_name (__main__.NamesTestCase)
Check combining and title()
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:/Users/silam/PycharmProjects/foundation/foundation.py", line 11, in test_first
    self.assertEqual(formatted_name, 'Yu2 Miao')
AssertionError: 'Yu Miao' != 'Yu2 Miao'
- Yu Miao
+ Yu2 Miao
?   +
```

```
----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (failures=1)
```

Above the "=====" line is an "F" which indicates failing occurred in the test file. And we can see exactly which test case method failed below the "=====" line. The docstring of that test method is also presented. The details of the error is also listed below.

12.1.3 Adding New Test

Now, let's expand the `get_formatted_name()` method so it also handles middle name case (so we have an additional case to test). The expanded method is as follows:

```
1 def get_formatted_name(first, last, middle=""):
2     """Generate a neatly formatted full name"""
3     if middle:
4         full_name = first + " " + middle + " " + last
5     else:
6         full_name = first + " " + last
7     return full_name.title()
```

we should add a new test case method to check cases that including middle name. To do this, we can add another method inside the the `NamesTestCase` class:

```
1 def test_first_last_middle_name(self):
2     """check name combining with middle name existing"""
3     formatted_name = get_formatted_name('yu', 'miao', 'super')
4     self.assertEqual(formatted_name, 'Yu Super Miao')
```

Now the test class `NamesTestCase` has two method, they can test different input cases. If test passed, the output should be:

```
..
----------------------------------------------------------------------
```

```
Ran 2 tests in 0.000s

OK
```

The complete test file is as follows:

```
import unittest
from name_function import get_formatted_name


class NamesTestCase(unittest.TestCase):
    """Test for 'name_function.py"""

    def test_first_last_name(self):
        """Check combining and title()"""
        formatted_name = get_formatted_name('yu', 'miao')
        self.assertEqual(formatted_name, 'Yu Miao')

    def test_first_last_middle_name(self):
        """check name combining with middle name existing"""
        formatted_name = get_formatted_name('yu', 'miao', 'super')
        self.assertEqual(formatted_name, 'Yu Super Miao')


unittest.main()
```

12.2 unittest.assert() Methods

The `assert()` methods in `unittest.TestCase` class contain more than `assertEqual()`. Following table shows commonly used assert methods that can be used to check the expected behavior of a function or class:

Method	Use
`assertEqual(a, b)`	test if `a == b`
`assertNotEqual(a, b)`	test if `a != b`
`assertTrue(x)`	test if `x` is `True`
`assertFalse(x)`	test if `x` is `False`
`assertIn(item, list)`	test if `item` is in `list`
`assertNotIn(item, list)`	test if `item` is not in `list`

12.3 Testing a Class

Testing a class is similar to testing a function. The main focus of testing a class is to test the behavior of the methods in the class.

We will use the following class as an example, it is defined in survey.py:

```
1  class AnonymousSurvey:
2      """Collect anonymous answers to a survey question."""
3
4      def __init__(self, question):
5          """Store a question, and prepare to store responses."""
6          self.question = question
7          self.responses = []
8
9      def show_question(self):
10         """Show the survey question."""
11         print(self.question)
12
13     def store_response(self, new_response):
14         """Store a single response to the survey."""
15         self.responses.append(new_response)
16
```

```
    def show_results(self):
        """Show all the responses that have been given."""
        print("Survey results:")
        for response in self.responses:
            print('- ' + response)
```

A driver program can be:

```
from survey import AnonymousSurvey

# Define a question, and use this question to make a survey
question = "What language did you first learn to code?"
my_survey = AnonymousSurvey(question)

# Show the question, and store the result of the question
while True:
    my_survey.show_question()
    answer = input("Input your answer, input q to quit: ")
    if answer == 'q':
        break
    else:
        my_survey.store_response(answer)

# show the survey result
print("Survey completed, thank you!")
my_survey.show_results()
```

Now, we want to write a test that verifies one aspect of the way `AnonymousSurvey` behaves: wether or not a single response to the survey question is stored properly. We'll use the `unittest.TestCase.assertIn()` method to verify that the response is in the list of responses after its been stored via `AnonymousSurvey.store_response()`. We are actually testing a method inside a class (like its member function), so it is similar with writting test

for function. The code is as follows:

```
import unittest
from survey import AnonymousSurvey


class SurveyTestCase(unittest.TestCase):
    """test for language survey"""

    def test_single_response_storage(self):
        """test if single response is stored properly"""
        survey = AnonymousSurvey("question")
        # try to store a string to the survey.responses list
        survey.store_response("dummy_response")
        # use assertIn() to verify storing of "dummy_response"
        self.assertIn("dummy_response", survey.responses)


# unittest.main()
```

(The code is named as test_survey.py)

Pay attention to the last line. In the textbook, this line is added. However, in my environment, I found that if I add this line, and I choose "Run Unittest in test_survey", I won't be able to run the test. Pycharm will show "No tests were found", and the console output is:

```
--------------------------------
Ran 0 tests in 0.000s


Launching unittests with arguments python -m unittest C:/Users/silam/PycharmProjects/fou
OK
```

To solve this problem, you have three ways.

1. don't choose "Run Unittest in test_survey", press alt+shift+F10 and choose run "test_survey"
2. delete `unittest.main()`, then "Run Unittest in test_survey"
3. add `if __name__ == '__main__':` before the `unittest.main()`

Reference for this problem can be found here.

12.3.1 The `setUp()` Method

When testing class's method, we have to create instances of the class. If we want to test different aspects of the class, we may have to create many different instances, which is a waste of time and memory space. The `unittest.TestCase` class has a `setUp()` method that allows you to create objects needed to test the class once and then use them in each of your test methods. When you include a `setUp()` method in a `TestCase` class (or its derived class), python runs the `setUp()` method before running each method starting with `test_`. Any objects created in the setUp()~ method are then available in each test method you write. This is like defining attributes in the test class. Actually, you define these object in the same way you define attributes in `__init__()` of a normal class, you have to use `self.object` to indicate this object is going to be used by all methods inside this class.

To continue the example of `AnonymousSurvey` class's test, we use `setUp()` to create object we need to test. The whole code is as follows:

```
1 import unittest
2 from survey import AnonymousSurvey
3
4
5 class SurveyTestCase(unittest.TestCase):
6     """test for language survey"""
7     def setUp(self):
8         """Create objects needed for testing:
9             - an AnonymousSurvey object
```

```
10              - a list containing three dummy responses
11          """
12          self.survey = AnonymousSurvey("dummy_question")
13          self.dummy_responses = ["response1", "response2", "response3"]
14
15      def test_single_response_storage(self):
16          # try to store strings in dummy_responses to the survey.responses
            ↪ list
17          for response in self.dummy_responses:
18              self.survey.store_response(response)
19          # use assertIn() to verify storing of "dummy_response"
20          for response in self.dummy_responses:
21              self.assertIn(response, self.survey.responses)
22
23
24  if __name__ == '__main__':
25      unittest.main()
```

Notice how to define objects for testing in the `setUp()` method.

12.3.2 Character Printed to Trace Testing

When a test case is running, Python prints one character for each unit test as it is completed. A passing test prints a dot, a test that results in an error prints an E, and a test that results in a failed assertion prints an F. This is why you'll see a different number of dots and characters on the frst line of output when you run your test cases. If a test case takes a long time to run because it contains many unit tests, you can watch these results to get a sense of how many tests are passing.

Chapter 13

place holder