

CS5001 Synthesis Assignment 1

Shang Xiao

February 25, 2022

Contents

1	Computational Thinking	3
1.1	Pattern Recognition	3
1.2	Abstraction	3
1.3	Decomposition	4
1.4	Algorithms	4
1.5	Flow Charts	6
2	Variables, Arithmetic Operations and Conditionals	7
2.1	Variables	7
2.2	Variable Types	8
2.3	Assignment Operators	9
2.4	Arithmetic Expressions	10
2.5	Concatenation	11
2.6	Boolean Expressions	12
2.7	Conditional Statements	14
2.8	Multiway Conditionals	15
2.9	Logical Operators	17
2.10	Tracing and Debugging	19
3	Functions and Testing	21
3.1	Making a Function	21
3.2	Calling a Function	23
3.3	Parameters	24
3.4	Global vs Local Scope	26
3.5	Return Statements	28
3.6	Testing Functions	29
4	While Loops	31
4.1	While Loops	31
4.2	Validating User Input	34
4.3	Debugging a Loop	35
5	Strings and Lists	36
5.1	Strings	36
5.2	Immutable Objects	37
5.3	Strings as Objects	39
5.4	General Sequences	40
5.5	Mutable Objects	41
5.6	Lists of Lists	43

1 Computational Thinking

1.1 Pattern Recognition

Explanation

To solve problems, we need to find patterns in words and actions by finding the similarities and differences between models. It is beneficial for us to improve our thinking to deal with complex problems.

Exercise

Give an example of pattern recognition.

Solution

As humans, we understand the world by recognizing the patterns around the world. We grow our understanding of language and society by observing words and actions to develop behavioural responses.

1.2 Abstraction

Explanation

Abstraction is the extraction of ideas from concrete things. Abstraction works by translating the phenomenon into a general view, which allows us to have a general idea of the problem and solve it. This process removes unnecessary details and patterns to help us better forming an opinion about a specific problem.

Exercise

Extract the general characteristics from the following recipe:

You need a bowl, a pair of chopsticks, a small oven, cake moulds, flour, eggs, sugar, and oil to make a cake. Pour four eggs into the bowl, stir them with chopsticks until you can see the bubble, put a spoon of sugar, continue stirring until the surface becomes thick, put another spoon of sugar, continue stirring, wait about 15 minutes until the eggs become creamy.

Solution

Abstraction	Specific Details
We know that each cake has its own ingredients	Flour, eggs, sugar, oil...
We know that each ingredient has a specified quantity	Pour four eggs into the bowl, then put two spoons of sugar...
We know that each cake needs a specified time to bake	Wait about 15 minutes until the eggs become creamy...

Table 1: Abstraction

1.3 Decomposition

Explanation

Complex problems can be simplified step by step. Decomposition means breaking down a complex problem into smaller pieces. Decomposition thinking is widespread in the computer science field. Usually, when we encounter large projects, the first thing we do is breaking the project into smaller tasks and then complete these tasks one by one, then complete the project.

Exercise

Give an example of the role of decomposition in learning Python.

Solution

When learning Python, we start with the basic syntax of the language. We then move to variables and conditionals, functions, lists, strings, etc..., then we know how to create and implement an algorithm. Lastly, we learn how to check errors in our code and make sure we can debug them. In this process, we took a step-by-step approach to expand our knowledge in Python.

1.4 Algorithms

Explanation

An algorithm refers to an accurate and complete description of a problem-solving scheme and a series of clear instructions to solve problems. An algorithm represents the strategy and mechanism to solve problems systematically. In other words, the required output can be obtained in a limited time for a given input specification. If an algorithm is flawed or inappropriate for a problem, executing the algorithm will not solve the problem. Algorithms vary by time complexity.

The instructions in an algorithm describe a computation process that starts with an initial state, passing through a finite and well-defined series of states, and eventually produce an output that stops in a final state.

Exercise

1. Briefly explain the role of an algorithm in problem solving.
2. Do algorithms always have structural elements?
3. Devise an algorithm that automatically calculates how many seconds have passed at any given time of the day.

Solution

1. In general, an algorithm returns a definite result through computation. This result can be used to solve practical problems.
2. No. While it is nice to have structural elements, they are not always required.

```
3. def main():
1     hours = int(input("Please enter the hours of the current time
2         with two digits, your input should be between 00-24"))
3     if hours >=0 and hours <=24:
4         print ("Valid Input")
5     else:
6         print("Invalid Input")
7
8     mins = int(input("Please enter the minutes of the current time
9         with two digits, your input should be between 00-60"))
10    if mins >=0 and mins <=60:
11        print ("Valid Input")
12    else:
13        print("Invalid Input")
14
15    secs = int(input("Please enter the seconds of the current time
16        with two digits, your input should be between 00-60"))
17    if secs >=0 and secs <=60:
18        print ("Valid Input")
19    else:
20        print("Invalid Input")
21
22    # calculations
23    if hours >= 0 and hours <= 24 and mins >= 0 and mins <= 60 and
24        secs>=0 and secs <=60:
25        print("Congratulations, your input is valid, please check
26            your answer below!")
27        hours_in_secs = hours * 3600
28        mins_in_secs = mins * 60
29        secs_in_secs = secs
```

```
25     secs_in_secs = secs
26     total_secs_today = hours_in_secs + mins_in_secs +
        secs_in_secs
27     print(total_secs_today, "seconds have passed since 00:00")
28
29 else:
30     print("Please verify if all three inputs are valid and try
        again")
31
32 main()
```

1.5 Flow Charts

Explanation

Flowcharts use rectangles, circles, diamonds, and possibly many other shapes to define the types of steps and connect arrows to restrict flows and sequences. They can be simple hand-drawn or comprehensive computer-generated diagrams depicting multiple steps and routes. If we consider all the different forms of flowcharts, they are among the most common diagrams on the planet, used in many fields by both technical and non-technical people.

Exercise

Draw a flowchart for an algorithm that automatically calculates how many seconds have passed at any given time of the day.

Solution

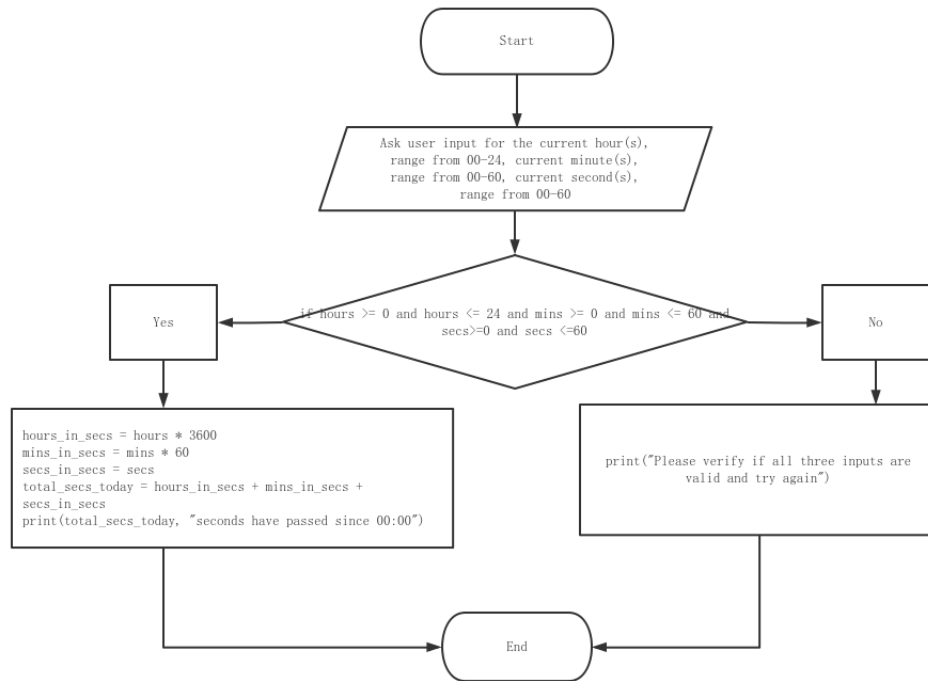


Figure 1: Flowchart for an algorithm that automatically calculates how many seconds have passed at any given time of the day.

2 Variables, Arithmetic Operations and Conditionals

2.1 Variables

Explanation

We often use variables to store data, to do this, we need to first assign variable names so that we can call them when we need them. Good variable names are easy to understand and should present the following characteristics:

- Must begin with a letter
- Can have both letters and numbers

- All lower-case
- Use underscores to separate words
- Nouns
- Avoid abbreviations

There are many types of variables. For example: all integers are type *int*, all decimals are type *float*, and all words are type *string*. We will be introducing these variable types in the next topic.

Exercise

Give a good variable name that describes odd number.

Solution

`odd_number`

2.2 Variable Types

Explanation

Like humans, computers have memories. A unique space is allocated in memory when a variable is created. Based on the variable's data type, the interpreter allocates specified memory and decides what data can be stored.

Type	Description
Numeric types	Numeric data types are used to store numeric values. They are immutable data types, which means that changing numeric data types allocates a new object. When we specify a value, the number object is created. Integer values, written as <i>int</i> , are whole numbers without fractional part and they can be positive, negative, or zero. In contrast, numbers that have a fractional part are known as <i>float</i> . <i>float</i> can also be positive, negative, or zero.
Literals	Literals are constant values assigned to constant variables. They cannot be modified after they are created.
Strings	Words are stored as <i>string</i> in Python. As one of the most commonly used data type in Python, <i>string</i> is a sequence of characters. We can use single or double quotes to delimit the ends of a <i>string</i> .
Checking variable types	If we are having trouble finding the correct variable types, we can use the <i>type()</i> function to determine the type. For example, 6 is a <i>int</i> , to check 6, write "type(6)" and will return <i>int</i>

Table 2: Variable types

Exercise

1. What variable type is "123"?
2. What variable type is 123?
3. What variable type is 12.3?

Solution

1. *string*
2. *int*
3. *float*

2.3 Assignment Operators

Explanation

Variable assignment in Python does not require a type declaration. Each variable is created in memory, including its identity, name, and data. Each variable must be assigned a value before it can be used, and then the variable

can be created. The equal sign $=$ is used to assign values to variables. The left side of the assignment operator is a variable name, and the right side of the assignment operator is the value stored in the variable.

Field	Description
Mathematics	$x = 12$ means the value of x is equal to twelve. In math, order does not matter. This means $x = 12$ is the same as $12 = x$.
Programming	$x = 12$ means x is set to twelve. Here, we usually refer x as the l-value and 12 as the r-value. In programming, order does matter. If we write $12 = x$, we will get a <code>SyntaxError: can't assign to literal</code> . This is because value cannot be on the left of an assignment operator.

Table 3: Assignment operators

Exercise

1. Assign number 1 to a variable called *num*.
2. Assign number 2 to the variable called *two*.

Solution

1.

```
num = 1
```
2.

```
two = 2
```

2.4 Arithmetic Expressions

Explanation

Arithmetic operators perform basic mathematical operations that output numerical values as a result. Arithmetic operators in programming have many similarities with arithmetic operations in mathematics, such as parentheses, exponent, multiplication, division, addition, and subtraction (known as PEMDAS).

Operator	Description	Associativity
()	Parentheses (grouping)	Left-to-right
**	Exponentiation	Right-to-left
+×, -×	Unary plus, unary minus	Left-to-right
*, /, //, %	Multiplication, division Integer division, remainder	Left-to-right
+, -	Addition, subtraction	Left-to-right

Table 4: Arithmetic expressions

Exercise

Calculate the following expressions:

1. $2^{**}3+8$
2. $3/2$
3. $3//2$
4. $2*3/2$
5. $20\%2^{**}4$

Solution

1. 16
2. 1.5
3. 1
4. 3.0
5. 4

2.5 Concatenation

Explanation

Sometimes we want to concatenate two strings together or create a *string* made of cycles of a substring. It turns out that two of the arithmetic operators, plus and multiplication, can be used to achieve these.

Operator	Description	Example
+	Concatenating two strings together.	"a"+"b" returns "ab"
*	One of the left or right variables is a string and the other is a number. Repeat a string multiple times.	"a"*4 or 4*"a" returns "aaaa"

Table 5: Concatenation

The "*" operator does concatenation only when one of the operands is an integer.

Exercise

Write the result of the following expressions:

1. "N"+"E"+"U"
2. "N"*4
3. "N"*4

Solution

1. NEU
2. NNNN
3. TypeError: can't multiply sequence by non-int of type 'str'

2.6 Boolean Expressions

Explanation

In Python, we can use Boolean expressions (also called predicates) to write different programs. There are two values of Boolean expression: *True* and *False*. Both are reserved words in Python. Some operators are also used within a Boolean expression that works similar to arithmetic operators, which we discussed earlier. These operators are called relational operators (comparison operators).

Operator	Description	Example
<	Less than	2<4
>	Greater than	4>2
<=	Less than or equal to	42<=42
>=	Greater than or equal to	44>=44
==	Equal to	2==2
!=	Not equal to	2!=4

Table 6: Boolean expressions

Relational operators have their precedence and association. So we need to check the order to calculate the result of the expression. It also turns out that *True* represents value 1 and *False* represents value 0 in Python. This means we can also use both *True* and *False* in arithmetic expressions.

Exercise

Check if the following expressions are correct.

1. 1<2

2. 1*3!=3*1

3. 1**2==2**0

4.

```

1 a = 3
2 b = 4
3 print(a*3+4<=b*3+3)
```

5. *True* < *False*

Solution

1. True

2. False

3. True

4. True

5. False

2.7 Conditional Statements

Explanation

The *if* statement is the first conditional statement that we will be introducing in this topic. In general form, *if* is a reserved word (decision word) usually followed by a Boolean expression (the condition) and a colon (indicates the start of the block). On the following line of a *if* statement, one or more indented statements (body of the *if*) are usually executed if the condition is true.

Sometimes, we want to execute different programs based on user input in Python. Unfortunately, the user input is not always correct. We need conditional statements to avoid problems before executing the program. We can use the *if* statement to check user inputs and parameters in Python. For example:

```
1 user_input = int(input("Please enter a integer"))
2 if user_input > 0:
3     print("True")
```

The program will print "True" as long as user enters a positive integer. Now you might be wondering: what happens when user enters a negative integer? In this case, the program will print nothing. And this is because we need another conditional statement called *else* for the situation when user does not follow instructions:

```
1 user_input = int(input("Please enter a positive integer:"))
2 if user_input > 0:
3     print("True")
4 else:
5     print("False")
```

Exercise

Write a program based on Figure 2.

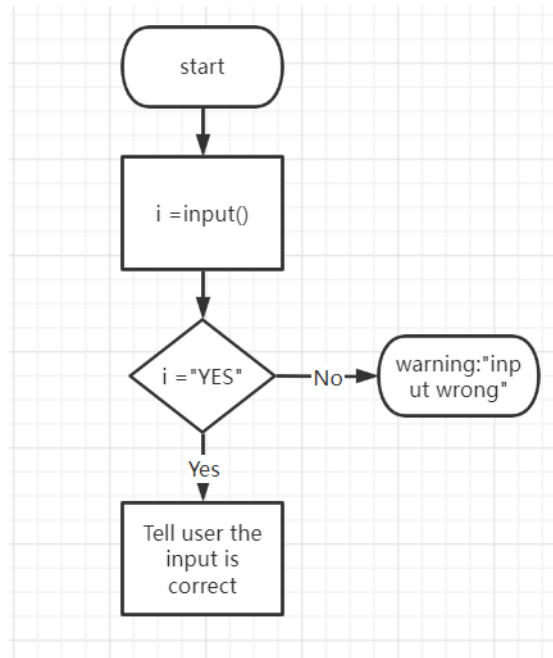


Figure 2: The *if* statement

Solution

```
1 i = input("Enter YES:")
2 if (i == "YES"):
3     print("True")
4 else:
5     print("False")
```

2.8 Multiway Conditionals

Explanation

Sometimes we just have too many conditions to check. The code will become long and ugly if we only use *if* and *else* to write programs. Luckily, Python provides us with another function called *elif*. We can use *elif* to make the code short and beautiful. Different from multiple *if* statements, *elif* improves the efficiency of programs when a condition meets one of the many statements:

```
1 s = input("Enter One or Two or Three or Four:")
2 if s == "One":
3     print(1)
4 elif s == "Two":
```

```

5     print(2)
6 elif s == "Three":
7     print(3)
8 elif s == "Four":
9     print(4)
10 else:
11     print ("Pls check your input and try again!")

```

Exercise

Write a program based on Figure 3.

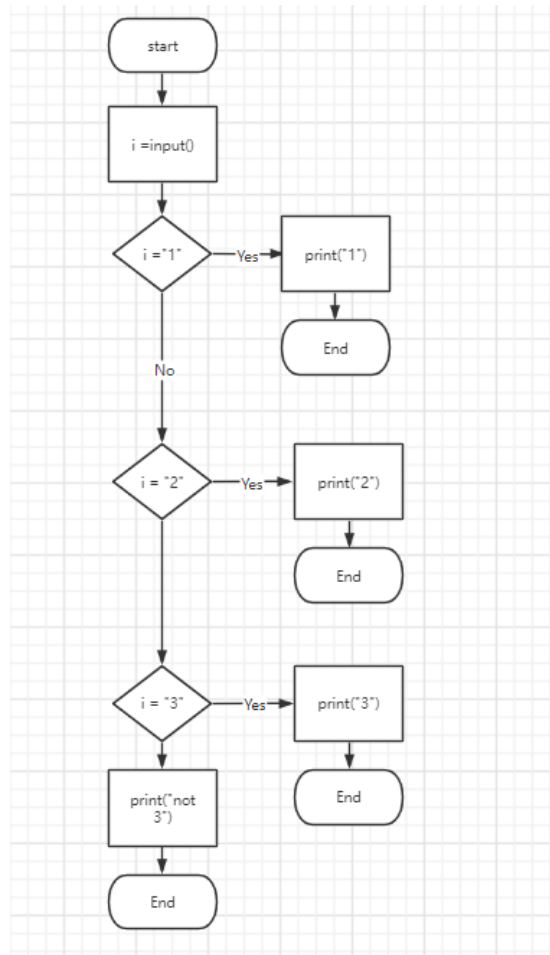


Figure 3: The *elif* statement

Solution

```
1 i = input("Please enter a integer:")
2 if i == "1":
3     print(i)
4 elif i == "2":
5     print(i)
6 elif i == "3":
7     print(i)
8 else:
9     print("Not 3")
```

The code can go even shorter, like this:

```
1 i = int(input("Please enter a integer:"))
2 if i>=1 and i<=3:
3     print(i)
4 else:
5     print("Not 3")
```

2.9 Logical Operators

Explanation

When encounter complex conditions, one statement simply cannot meet our needs. In Python, we can use logical operators to combine one or more simple Boolean expressions together. There are three logical operators: *and*, *or*, *not*.

Operator	Description
<i>and</i>	A <i>and</i> B determines the result of the expression collaboratively. If A is true, then B determines the result of A <i>and</i> B and returns B. If A is false, then A determines the result of A <i>and</i> B and returns false. Keep note that Python will not continue evaluating B after A has been determined false because false <i>and</i> anything is always false. This is called "short-circuit evaluation".
<i>or</i>	A <i>or</i> B in Python is referred as "inclusive <i>or</i> ", this means, A <i>or</i> B is only false when both A and B are false, which is different from "exclusive <i>or</i> " in casual conversions. For example: when someone asking if you want apple or orange, they mean "exclusive <i>or</i> ", the implication is that you can have either apple or orange but not both. In Python it is not. Keep note that Python will not continue evaluating B after A has been determined true because true <i>or</i> anything is always true.
<i>not</i>	<i>not</i> means negation. <i>not</i> A returns the reversed value of A. If A is true, then <i>not</i> A returns false, and vice versa. Keep note that <i>not</i> (A==B) is the same as A!=B, <i>not</i> (A!=B) is the same as A==B, and <i>not</i> (A<B) is the same as A>=B, do not forget to put the equal sign here!

Table 7: Logical operators

Exercise

1. Which logical operator is TB-1 representing?

A	B	Result
True	True	True
True	False	False
False	True	False
False	False	False

Table 8: TB-1

2. Which logical operator is TB-2 representing?

A	B	Result
True	True	True
True	False	True
False	True	True
False	False	False

Table 9: TB-2

Solution

1. *and*
2. *or*

2.10 Tracing and Debugging

Explanation

When solving a problem, we should always analyze the question first to understand what the question is really asking. Before writing the actual code, we need to make sure to have a well-presented flowchart to guide our logic. We should follow the flow of the flowchart to make sure we can write clear, organized codes. After completing the code, we should test the program with some test cases to prove the correctness of our program.

Exercise

Devise a *main()* function for the following code to check if all tests has been passed.

```

1 def euclidean(x1, y1, x2, y2):
2     a=(x2 - x1) ** 2
3     b=(y2 - y1) ** 2
4     c=(a + b) ** 0.5
5     return (round(c,3))
6
7 def test_euclidean(x1, y1, x2, y2, e):
8     actual = euclidean(x1, y1, x2, y2)
9     return (x1, y1, x2, y2, e)
10
11 def exp1():
12     e1 = float(input("1st Expected:"))
13     a = int(input("1st X1"))
14     b = int(input("1st Y1"))
15     c = int(input("1st X2"))
16     d = int(input("1st Y2"))
17     return (a,b,c,d,e1)
18
19 def exp2():

```

```

20     e2 = float(input("2nd Expected:"))
21     aa = int(input("2nd X1"))
22     bb = int(input("2nd Y1"))
23     cc = int(input("2nd X2"))
24     dd = int(input("2nd Y2"))
25     return (aa, bb, cc, dd, e2)
26
27 def exp3():
28     e3 = float(input("3rd Expected:"))
29     aaa = int(input("3rd X1"))
30     bbb = int(input("3rd Y1"))
31     ccc = int(input("3rd X2"))
32     ddd = int(input("3rd Y2"))
33     return (aaa, bbb, ccc, ddd, e3)
34
35 def exp4():
36     e4 = float(input("4th Expected:"))
37     aaa = float(input("4th X1"))
38     bbb = float(input("4th Y1"))
39     ccc = float(input("4th X2"))
40     ddd = float(input("4th Y2"))
41     return (aaa, bbb, ccc, ddd, e4)
42
43 def equ():
44     fail = 0
45     if exp1() == run1():
46         print("PASSED")
47     else:
48         print("FAILED")
49         fail += 1
50     if exp2() == run2():
51         print("PASSED")
52     else:
53         print("FAILED")
54         fail += 1
55     if exp3() == run3():
56         print("PASSED")
57     else:
58         print('FAILED')
59         fail += 1
60     if exp4() == run4():
61         print("PASSED")
62     else:
63         print("FAILED")
64         fail += 1
65     return fail
66
67 def run1():
68     test_euclidean(0,0,0,0,0.0)
69     return (test_euclidean(0,0,0,0,0.0))

```

```

70 def run2():
71     test_euclidean(2,-1,-2,2,5.0)
72     return (test_euclidean(2,-1,-2,2,5.0))
73 def run3():
74     test_euclidean(0,0,1,1,1.4142135623730951)
75     return(test_euclidean(0,0,1,1,1.4142135623730951))
76 def run4():
77     test_euclidean(-5.2,3.8,-13.4,0.2,8.955445270895243)
78     return(test_euclidean(-5.2,3.8,-13.4,0.2,8.955445270895243))

```

Solution

```

1 def main():
2     failure = equ()
3     if failure == 0:
4         print ("FINAL TESTING...")
5         print ("FINAL RESULT: ALL TESTS PASSED")
6     else:
7         print ("FINAL TESTING...")
8         print ("FINAL RESULT: TEST FAILED, Please validate your input
          and try again")
9 main()

```

3 Functions and Testing

3.1 Making a Function

Explanation

So far, we have learned how to solve problems with all the basic tools we need in Python. From this topic, we will be looking into more advanced tools to solve problems well. Beginning with functions.

A function is mini-program, an organized, reusable piece of code that is executed more than once, and completes one job. Python provides us with many built-in functions in its library such as *print()*. We can also create our own functions, which are called user-defined functions.

We can define a function that has its own desired feature. Here are some simple rules to get started:

- The function code block begins with the *def* keyword, followed by the function identifier name and parentheses ().
- Parameters must be placed between parentheses.
- The first line of the function can optionally use strings to hold the function description.

- Function content starts with a colon and is indented.
- The *return* statement terminates a function, returning a value to user. *return* without an expression is equivalent to returning *None*.

Characteristic	Description
Name	A unique identifier for every function. The name of a function should be descriptive and easy to understand. Usually, we use a verb to express the purpose of a function.
Parameters	Functions take inputs, also called parameters. If we were to write a function that calculates the product of two integers, then the two integers are its arguments.
Return type	The value that the function is responsible for. If we were to write a function that calculates the product of two floats, then the function will return us the product of these two floats, which is a number (<i>float</i>).

Table 10: Making a function

To define a function in Python, use the *def* statement followed by the function name, parentheses, arguments in parentheses, and colons. Remember, body of the function should be written in an indented block, and the value of the function is returned by the *return* statement.

Exercise

1. Write a function called *my_abs()* that takes an argument *x* and return its absolute value.
2. Write a function called *collatz()* that takes an argument called *number*. If the argument is even, then *collatz()* prints *number/2*. If *number* is odd, *collatz()* prints *3*number+1*.

Solution

```

1. _____
1  def my_abs(x):
2      if x >= 0:
3          return x
4  else:
5      return -x

```

```
2.
1 def collatz(number):
2     print number // 2 if number % 2 == 0 else 3 * number + 1
```

3.2 Calling a Function

Explanation

Python has many useful built-in functions that we can call directly. To call a function, we need to know the function's name and its arguments, such as *abs()*, the absolute value function, which has only one argument.

We can view official Python documentation from the following website:

<http://docs.Python.org/3/library/functions.html>

We can use command line to call *abs()* function:

```
1 >>> abs(100)
2 100
3 >>> abs(-20)
4 20
5 >>> abs(12.34)
6 12.34
```

TypeError is returned once the user enters a wrong number of arguments. In the following case, Python explicitly tells us that *abs()* only takes one argument, but was given two:

```
1 >>> abs(1, 2)
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   TypeError: abs() takes exactly one argument (2 given)
```

Other functions such as *max()* can take more than one argument. For example:

```
1 >>> max(1, 2)
2 2
3 >>> max(2, 3, 1, -5)
4 3
```

If the number of arguments are correct but the type of arguments are not accepted by the function, *TypeError* is also returned:

```
1 >>> abs('a')
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   TypeError: bad operand type for abs(): 'str'
```

Exercise

1. Call a function that reads a value.
2. Call a function that converts a value to its absolute value.
3. Call a function that converts a value to its square root value.

Solution

```
1. _____
1  >>> input()
2  12
3  '12'
_____

2. _____
1  >>> abs(int(input()))
2  12
3  12
4  >>> abs(int(input()))
5  -3
6  3
_____

3. _____
1  >>> import math
2  >>> math.sqrt(int(input()))
3  4
4  2.0
_____
```

3.3 Parameters

Explanation

Parameters are variables. Values assigned to a parameter are called arguments. To understand how parameters and arguments are stored in our computers, we need to look at the execution stack, a chunk of memory assigned just for our program. Once a chunk of memory has been assigned, no other part of the computer can access it. When we define a variable in our *main()* function, the variable is stored at the bottom of the execution stack.

Then, when we invoke our function with an argument value assigned to the variable, a higher level of data is stored in the execution stack.

Lastly, whatever is left in the function will then be pushed into the highest level of the execution stack. Once we hit enter, the result is returned to us, and all of the data stored in these three levels are removed permanently.

Name	Description
Default arguments	Python allows function arguments to have default value.
Keyword arguments	Python allows functions to be called using keyword arguments.
Arbitrary arguments	Python allows us to handle some situation through function calls with an arbitrary number of arguments.

Table 11: Arguments

When defining a function, we can assign a default value to the parameter, such as `df=1`. If no value is eventually assigned to a parameter, the default value is used:

```

1 # Correct way to define default parameters: positional argument comes
  # first, default argument comes later
2 def print_hello(name, df=1):
3     print (name, df)
4 # Wrong definition
5 def print_hello(df=1, name):
6     print (name, df)
7 # When called without assigning the value of df, the default value is 1
8 def main():
9     print_hello('Jim')
10 main()
11 # When called with the value of df assigned and specified as 2
12 def main():
13     print_hello('Jim',2)
14 main()

```

Functions that are specified in keyword form makes functions easier to use while eliminating the need for arguments to be ordered:

```

1 # Here is an example of a function being called correctly with keyword
  # arguments
2 # print_hello('Jay', sex=1)
3 # print_hello(1, name='Jay')
4 # print_hello(name='Jay', sex=1)
5 # print_hello(sex=1, name='Jay')
6
7 # The following is the wrong call
8 # print_hello(name='Jay', 1)
9 # print_hello(sex=1, 'Jay')

```

When there are positional arguments, they must precede keyword arguments. There is no order between keyword arguments.

Exercise

Given a set of numbers A, B, C, \dots , calculate $A^2 + B^2 + C^2 + \dots$

Solution

To define a function for this question, we need to determine its parameters. Since the number of parameters is uncertain, we can think of A, B, C, \dots as a list or tuple. The function can be defined as follows:

```
1 def calc(numbers):
2     sum = 0
3     for n in numbers:
4         sum = sum + n * n
5     return sum
```

To call the function, we need to create a list or tuple:

```
1 >>> calc([1, 2, 3])
2 14
3 >>> calc((1, 3, 5, 7))
4 84
```

With mutable arguments, the method of calling a function can be simplified as follows:

```
1 >>> calc(1, 2, 3)
2 14
3 >>> calc(1, 3, 5, 7)
4 84
```

We need to change the parameters of the function to mutable parameters:

```
1 def calc(*numbers):
2     sum = 0
3     for n in numbers:
4         sum = sum + n * n
5     return sum
```

3.4 Global vs Local Scope

Explanation

Global variable scope prevents the creation of a local copy of a variable. Local variable scope refers to any variable defined in the body of a function, including the parameters. Variables commonly assigned are called local variables, and variables defined outside a local environment are called global variables. Global variables do not have any indentation and can be called from anywhere.

Name	Description
Scope	Visible range of an identifier. Often referred as the scope of a variable.
Global scope	Visible throughout the program environment.
Local scope	Visible inside a function, class, etc. Local variables cannot be used beyond their local scope.

Table 12: Global vs local scope

Consider the following example:

```

1 ID = "A"
2 def changeID():
3     ID = "B"
4     print("changeID", ID)
5 changeID()
6 print(ID)

```

Output:

```

1 change_NAME B
2 A

```

Before defining the *changeID()* function, we assigned global variable *ID* with value of A. After defining the *changeID()* function, we assigned local variable *ID* with value of B.

At this point, local variable *ID* assigned with value of B within the *changeID()* function has been given the same name as the global variable *ID* assigned with value of A outside the *changeID()* function.

When *changeID()* is called, local variable *ID* with value of B takes precedence over the global variable *ID* with value of A outside function *changeID()*.

When *print(ID)* is executed, global variable *ID* with value of A outside function *changeID()* takes effect.

If there are no global variables, local variables are executed first, and vice versa.

Another example:

```

1 name = ["pony", "jack"]
2 print(1, name)
3 def change_name():
4     name = "nicholas"
5     print("change_name", name)
6 change_name()
7 print(2, name)

```

Output:

```
1 ['pony', 'jack']
2 change_name nicholas
3 2 ['pony', 'jack']
```

If a function has the *global* keyword, its variable is essentially a global variable that can be read and assigned.

Exercise

Analyze the following code and its output.

```
1 NAME = "nicholas"
2 print(1, NAME)
3 def change_NAME():
4     global NAME
5     NAME = "niubi"
6     print("change_NAME", NAME)
7 change_NAME()
8 print(2, NAME)}
```

Solution

NAME = "*nicholas*" acts globally.

When *print*("1", *NAME*) is executed, *NAME* uses the global variable.

When *change_NAME*() is executed with keyword *global*, *NAME* = "*nicholas*" will be changed to global variable *NAME* = "*niubi*"

When *print*("2", *NAME*) is executed, since we executed *change_NAME*() using a global keyword, *print*("2", *NAME*) will now return *niubi*.

```
1 nicholas
2 change_NAME niubi
3 2 niubi
```

3.5 Return Statements

Explanation

The *return* statement ends functions and hands back execution result from the invoked location. The *return* statement can return any object.

return is for computers, *print* is for humans.

We can omit the function's return value and return no value. We can also omit the entire return statement. In both cases, the return value is *None*.

There are two kinds of *return* statements:

Name	Description
Explicit <i>return</i>	An explicit <i>return</i> statement immediately terminates a function's execution and sends the return value back to the user.
Implicit <i>return</i>	If we do not use the return value explicitly in the <i>return</i> statement, or if we omit the <i>return</i> statement entirely, Python will implicitly return a default value for us. The default return value will always be <i>None</i> .

Table 13: *return* statements

In interactive mode, result of a *return* statement is automatically printed. Most of the time, we need the *print* function to display the result.

Exercise

1. True or false: *print* and *return* are the same thing and can be used interchangeably.
2. What will a function return if it has no *return* statement?
3. True or false: *return* allows any code remaining in the function to be executed.

Solution

1. False. In interactive mode, the result of *return* is automatically printed, but when running as a code alone, the *print* function is required to display the result.
2. If no value is explicitly returned, *None* is returned.
3. False. Upon *return*, the program terminates.

3.6 Testing Functions

Explanation

Testing function is an important step when writing codes. Before adding our functions to a larger program, we should double-check the correctness of these functions. To do this, we need to create a new test file, usually we can name it *xx_test.py*:

- Create a new test file called *xx_test.py*.
- Import the function or class that we want to test. For example:

```
1 from distance import euclidean
```

- Define our test function. We can name it *test_xx()*. The test function should include all of the parameters in the original function and an extra parameter called the expected value, which has been computed before comparing with the actual value. Note that the expected and actual values may not be the same every time, and this is when we need to go back and check our functions.
- Define another function called *run_xx_tests()*. This function only does one thing: calling *test_xx()* and print its result.
- Lastly, call *run_xx_tests()* in *main()*.

Exercise

1. Write a function to compute the area of a rectangle with length *a* and width *b*.
2. Write a testing function to test the correctness of function in Exercise 1.

Solution

```

1. _____
1  def area(a,b):
2  ''' Function area
3  Parameters: two integers, representing length and width
4  Returns: a integers, the area of the rectangle
5  '''
6  return a*b
_____

2. _____
1
2  from area import area
3
4  EPSILON = 0.001
5
6  def test_area(a, b, expected):
7
8  print('Testing length : '+str(a)+'width : '+str(b))
9  actual = area(a, b)
10 return abs(actual - expected) < EPSILON
11
12 def run_area_tests():
13 ''' function run_area_tests
14 Parameters: none
15 Returns: an int, number of tests that failed
16 '''
17 num_fail = 0
18
19 # Test 1: a:0, b:0 Expected: 0
20 if (test_area(0, 0, 0)):

```

```

21 print('PASSED! :)')
22 else:
23 print('FAILED :(')
24 num_fail += 1
25
26 # Test 2: a:2, b:3. Expected: 6
27 if (test_area(2, 3, 6)):
28 print('PASSED! :)')
29 else:
30 print('FAILED :(')
31 num_fail += 1
32
33 # Test 3: a:1.2, b:1.2. Expected: 1.44
34 if (test_area(1.2, 1.2, 1.44)):
35 print('PASSED! :)')
36 else:
37 print('FAILED :(')
38 num_fail += 1
39
40 # Test 4: a:100, b:1000. Expected: 100000
41 if (test_area(100, 1000, 100000)):
42 print('PASSED! :)')
43 else:
44 print('FAILED :(')
45 num_fail += 1
46
47 return num_fail
48
49
50 def main():
51 print('Testing area functions...\n\n')
52 failures = run_area_tests()
53 if failures == 0:
54 print('Everything passed, great job functions!')
55 else:
56 print('Something went wrong, go back and fix pls.')
57
58 main()

```

4 While Loops

4.1 While Loops

Explanation

Loops are used to execute blocks repeatedly. We have learned how to use indentation to indicate block membership in our codes. Loops are written similarly.

Python has two primitive loop commands:

- *while* loops
- *for* loops

We can execute a set of statements with a *while* loop as long as the condition is properly stated. When conditions are not met, the program will not execute the stuff inside the loop. For example:

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

For every loop command we have written, make sure that there is a way to exit the loop. One way is by using the *break* statement:

```
1 i = 4
2 while i < 10:
3     print(i)
4     if i & 1:
5         break
6     i += 1
```

With the *continue* statement, we can stop the current iteration and proceed to the next iteration:

```
1 i = 0
2 while i < 6:
3     i += 1
4     if i == 3:
5         continue
6     print(i)
```

Use the *else* statement to run the code block once the condition is no longer met:

```
1 i = 1
2 while i < 6:
3     print(i)
4     i += 1
5 else:
6     print("i is no longer less than 6")
```

For loop allows us to iterate over a list, performing the same action on each item in the list.

Exercise

1. Write a program to decide if a word is a palindrome. (My solution here is different from any of the solutions discussed during the lecture)
2. Take the value of n and find n factorial.
3. Print out the first 100 Fibonacci numbers.

Solution

```
1. 

---

1 def reverse(x):  
2     # myString = "Hello"  
3     location = -1  
4     while(location >= -len(x)):  
5         return(x[location])  
6         location -=1  
7 def normal(x):  
8     location = 0  
9     while(location < len(x)):  
10        return(x[location])  
11        location +=1  
12 def main():  
13     x = input("Enter a word:")  
14     if normal(x) == reverse(x):  
15         print("Your word is palindrome!")  
16     else:  
17         print("Sorry, your word is not palindrome!")  
18  
19 main()  
2. 

---

1 n = int(input())  
2 for i in range(1,n+1):  
3     s=s*i  
3. 

---

1 a = 1  
2 b = 1  
3 for i in range(1,100):  
4     print(a, " ")  
5     tmp = a+b  
6     a = b  
7     b = tmp  
4. 

---


```

4.2 Validating User Input

Explanation

Validation is performed to ensure that only adequately formed data enters the program to prevent incorrectly formed data from persisting in the background and triggering failures. Therefore, input validation should occur as early as possible in the data flow, preferably as soon as received externally.

We can use *while* loop to check if the data entered is valid. If data is valid, proceed to the next step; if data is invalid, prompt the user to enter again.

Exercise

1. Write a program to validate user input is a digit.
2. Write a program to validate user input is a digit between 0-10.
3. The expected user input for the program below is?

```
1 import math
2 def main():
3     value = int(input("Please enter a positive number:\n"))
4     while value <= 0:
5         value = int(input("Try again!\n"))
6     print("Thank you!")
7
8 main()
```

Solution

1.

```
1 import math
2 def main():
3     value = input("Please enter a digit:\n")
4     while not(value.isdigit()):
5         value = input("Not a digit. Try again!\n")
6     print("Thank you!")
7
8 main()
```

2.

```
1 import math
2 def main():
3     value = int(input("Please enter a digit between 0-10:\n"))
4     while value > 10 or value < 0:
5         value = int(input("Not a digit between 0-10. Try again!\n"))
6     print("Thank you!")
7
8 main()
```

3. A value greater than zero.

4.3 Debugging a Loop

Explanation

When debugging a loop, we can use the *print* function. The Python debugger also provides a debugging environment for Python programs. It supports setting conditional breakpoints, executing source code line by line, stack checking, and more.

Using *print* statement to debug:

```
1 i = 9
2 while i > 0:
3     print(i)
4     i -= 1
```

Using PyCharm (IDE) debugger to debug:

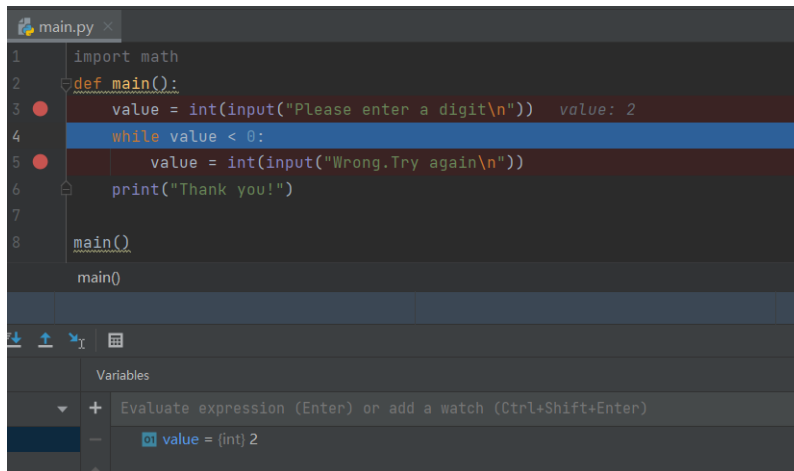


Figure 4: Debugging in PyCharm

Exercise

1. Debug the following program using *print* statement so it starts counting from 1.

```
1 def main():
2     i = 0
3     sm = 0
4     while i<10:
5         sm+=i
```

```
6         i+=1
7     main()
```

2. Explain how a debugger works.

Solution

1. By adding the *print* statement, we can see that *i* was initially set to 0, but we want the program to start from 1:

```
1 def main():
2     i = 0
3     sm = 0
4     while i<10:
5         print(i)
6         sm+=i
7         i+=1
8 main()
```

Based on our finding, we can make the following correction:

```
1 def main():
2     i = 1
3     sm = 0
4     while i<10:
5         print(i)
6         sm+=i
7         i+=1
8 main()
```

2. A debugger is a program that helps detect and correct program errors. A debugger uses breakpoints to stop the execution of the program in order to check the value of the variable at that point.

5 Strings and Lists

5.1 Strings

Explanation

A string is an array of characters represented by Unicode. Strings can be created by enclosing characters in single or double quotations. Triple quotes can even be used in Python but commonly represent multi-line strings.

We can use the index method to access individual characters in a string. An index allows positive and negative addresses to reference characters in a string, such as 0 for the first character and -1 for the last character. An `IndexError` is

returned when an out-of-scope index is accessed. Only integers can be assigned as indexes.

Exercise

1. Enter a string and print its character in reverse order.
2. Given a string, count the number of character 'a' in the string.

Solution

```
1. 

---

1 s = "this is a string"
2 i = len(s)-1
3 while ~i:
4     print(s[i])
5     i-=1


---


2. 

---

1 s = "this is a string"
2 cnt = 0
3 for e in s:
4     if e == 'a':
5         cnt+=1
6 print(cnt)


---


```

5.2 Immutable Objects

Explanation

Objects allocated by Python fall into two categories: mutable objects and immutable objects. To have a mutable object means the content of an object is mutable (modifiable), and to have an immutable object means the content of an object is immutable (non-modifiable).

Name	Example
Immutable objects	<i>int, string, float, number, tuple</i>
Mutable objects	<i>list, dictionary</i>

Table 14: Immutable and mutable objects

Strings are immutable objects and cannot be modified in place. If we want to change the value of a string, we can create a new string object. If we modify any elements in the string, we will get `TypeError`:

```

>>> s = "hello"
>>> s[0] = 'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>>

```

Figure 5: Modify a string

Although we cannot modify a string once created, we can assign a new variable to create another string if we wish:

```

1 s = "this is a string"
2 s = "hello world"
3 print(s)

```

In the example below, the program will first execute line 8 in *main()* and print "Before calling change, x= B" because at this time the local variable *x* inside *main()* is assigned to value "B".

On line 9, we called *change(x)*, within *change(x)*, on line 2, the function will execute whatever variable was assigned to local variable *x* in *main()* so it prints "At start of change, x= B".

On line 3, we change the parameter variable *x* value inside *change(x)* to "A", it prints "At end of change, x= A".

On line 10, because *x* is immutable; inside *change(x)*, it was the formal parameter variable called *x* that was changed, not the local variable *x* inside *main()*, thus it will print "After calling change, x= B"

```

1 def change(x):
2     print("At start of change, x=", x)
3     x = "A"
4     print("At end of change, x=", x)
5
6 def main():
7     x = "B"
8     print("Before calling change, x=", x)
9     change(x)
10    print("After calling change, x=", x)
11
12 main()

```

Output:

```

1 Before calling change, x= B
2 At start of change, x= B
3 At end of change, x= A
4 After calling change, x= B

```

Exercise

How many strings are created in the code below?

```
1 first = "Python"
2 last = "3.7"
3 print("My language is" + " " + first + " " + last)
```

Solution

Strings are immutable. A new string is created every time we concatenate. The seven strings are: "Python", "3.7", "My language is", " "My language is ", "My language is Python", "My language is Python ", "My language is Python 3.7".

5.3 Strings as Objects

Explanation

Object-oriented thinking is a transformative way of thinking, a philosophy leading the progress of programming. If we do not want to do things redundantly, it is better to plan ahead with the thought of creating objects in the beginning. In object-oriented programming, we write modular and easily reusable codes when defining a class that creates a new object represented by a data set.

Everything in Python is a class, such as *int*, *float*, and *string*. Objects of a class have their own unique data as the values for that instance.

String is the class that represents strings in Python. When we learned how to write functions, we use the functions that we have written to do tasks repeatedly. When a function is put inside of a class, the function operates as a method of that class. Since a method is a part of a class, in order to call a method, we need to know its name and parameters. We can call methods using the dot notation:

```
1 object.method_name(parameter_list)
```

For example, the *capitalize()* method used on an old string returns a new string with the first letter capitalized and rest lower-cased:

```
1 x = "left4dead"
2 x.capitalize() = "Left4dead"
```

The *upper()* method used on an old string returns new string with all upper-cases:

```
1 a = 'csgo'
2 a.upper() = 'CSGO'
```

The *capitalized* and *upper* methods works on an old string with a combination of lower-cased and upper-cased characters together as well:

```

1 x = "AbcDDeFFgrR"
2 x.capitalize() = "AbcddeffgrR"
3
4 a = 'AbcDDeFFgrR'
5 a.upper() = 'ABCDEFfGRR'

```

Exercise

What is the class of *nu*?

```

1 nu = "university"

```

Solution

str

5.4 General Sequences

Explanation

Strings are not the only sequences we have. There many other general sequences in Python such as *tuple* and *list*.

Category	Tuple	List
Delimits by	Parentheses	Square brackets
Have order	Yes	Yes
Modification	Immutable	Mutable
Element uniqueness	Not unique	Not unique
Element access	The index can be accessed in the same way as the <i>list</i> . The contents of the index cannot be changed, but if a <i>list</i> is used as an element of a <i>tuple</i> , you can change the <i>tuple</i> by changing the <i>list</i>	<i>list</i> accesses content by numeric index starting at 0. <i>list</i> can also be accessed in reverse order using a negative index starting at -1

Table 15: General sequences

Exercise

1. Create a tuple object which contains many strings. Count the number of the first string in a tuple using the correct function.

2. Create a list object which contains many strings. Transfer the first string to the end.

Solution

```
1. 

---

1 tp = ("a","b","c","a","ab")  
2 print(tp.count(tp[0]))

---


```

```
2. 

---

1 lst = ["a","b","c"]  
2 str = lst[0]  
3 lst.pop(0)  
4 lst.append(str)  
5 print(lst)

---


```

5.5 Mutable Objects

Explanation

In contrast to immutable objects, we can change the contents within any mutable objects. When mutable objects are passed to a function called by the user, they behave similarly to immutable objects when we modify what the parameter variable is referring to. But if the function modifies the object, the changes can also be seen by the caller. If we want to change the objects to a new object, we can use the function *copy()* to create another object with the same content as the old one.

In the example below, the program will first execute line 8 in *main()* and print "Before calling change, z= [4, 5, 6]" because at this time the local variable *z* inside *main()* is assigned to list [4, 5, 6].

On line 9, we called *change(z)*, within *change(z)*, on line 2, the function will execute whatever variable was assigned to local variable *z* in *main()* so it prints "At start of change, z= [4, 5, 6]".

On line 3, we change the parameter variable *z* value inside *change(z)* to list [1, 2, 3], it prints "At end of change, z= [1, 2, 3]".

On line 10, because *z* is mutable; inside *change(z)*, it was the formal parameter variable called *z* that was changed, not the local variable *z* inside *main()*, thus it will print "After calling change, z= [4, 5, 6]"

```


---

1 def change(z):  
2     print("At start of change, z=", z)  
3     z = [1, 2, 3]  
4     print("At end of change, z=", z)  
5  
6 def main():  
7     z = [4, 5, 6]  
8     print("Before calling change, z=", z)
```

```

9     change(z)
10    print("After calling change, z=", z)
11
12 main()

```

Output:

```

1 Before calling change, z= [4, 5, 6]
2 At start of change, z= [4, 5, 6]
3 At end of change, z= [1, 2, 3]
4 After calling change, z= [4, 5, 6]

```

But this time, we want to modify the formal parameter variable value, rather than changing the value it is referring to:

```

1 def modify(z):
2     print("At start of modify, z=", z)
3     z[0] = 8
4     print("At end of modify, z=", z)
5
6 def main():
7     z = [4, 5, 6]
8     print("Before calling modify, z=", z)
9     modify(z)
10    print("After calling modify, z=", z)
11
12 main()

```

Output:

```

1 Before calling modify, z= [4, 5, 6]
2 At start of modify, z= [4, 5, 6]
3 At end of modify, z= [8, 5, 6]
4 After calling modify, z= [8, 5, 6]

```

What do we see this time? The last line of the output has also changed. This is exactly what happened to the mutable object *z* within the *modify(z)* function:

The program will first execute line 8 in *main()* and print "Before calling modify, z= [4, 5, 6]" because at this time the local variable *z* inside *main()* is assigned to list [4, 5, 6].

On line 9, we called *modify(z)*, within *modify(z)*, on line 2, the function will execute whatever variable was assigned to local variable *z* in *main()* so it prints "At start of modify, z= [4, 5, 6]".

On line 3, we modify the parameter variable *z* value inside *change(z)* to *z[0] = 8*, it prints "At end of modify, z= [8, 5, 6]".

On line 10, because *z* is mutable; inside *change(z)*, it was the formal parameter variable called *z* that was modified, but since list is a mutable object, the

modification we did to formal parameter variable z will also reflect on the local variable z inside *main()*, thus it will print "After calling modify, $z = [8, 5, 6]$ "

Exercise

1. Analyze the following program:

```
1 def change(a):
2     a.append(1)
3
4 b = (1,2,3)
5 change(b)
6 print(b)
```

2. Analyze the following program:

```
1 def change(a):
2     a.append(5)
3
4 def main():
5     data = [1,2,3,4]
6     copy = data.copy()
7     change(copy)
8     print(data)
9     print(copy)
10
11 main()
```

Solution

1. Since tuple is an immutable object, we will get an error.
2. Because *copy()* will create a new object, *copy* and *data* will not effect each other. The function will change the content of *copy*. So the result is:

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

5.6 Lists of Lists

Explanation

When we want to write a matrix in Python, we can use lists of lists. We can put a list into another list to form a matrix where the first index represents the

row and the second index represents the column. For example:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

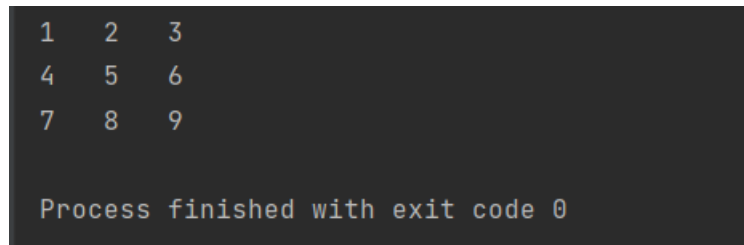
We can declare a list of list like this:

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

We can display a list in a form of mathematical matrix using *while* loop:

```
1 matrix = [  
2     [1,2,3],  
3     [4,5,6],  
4     [7,8,9]  
5 ]  
6  
7 row = 0  
8 while row<len(matrix):  
9     column = 0  
10    while column<len(matrix[row]):  
11        print(matrix[row][column],end='\t')  
12        column+=1  
13    print()  
14    row+=1  
15 column = 0
```

Output:



```
1  2  3  
4  5  6  
7  8  9  
  
Process finished with exit code 0
```

Figure 6: Matrix

Exercise

Write a program to store and display the following matrix:

$$\begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \end{pmatrix}$$

Solution

```
1 matrix = [  
2     ['a','b','c','d'],  
3     ['e','f','g','h'],  
4     ['i','j','k','l']  
5 ]  
6  
7 row = 0  
8 while row<len(matrix):  
9     column = 0  
10    while column<len(matrix[row]):  
11        print(matrix[row][column],end='\t')  
12        column+=1  
13    print()  
14    row+=1  
15 column = 0
```
