

Subroutine

01204111 Section 1

```
import codecs
import os
import re
import sys

from setuptools import setup, find_packages

here = os.path.abspath(os.path.dirname(__file__))

def read(*parts):
    # intentionally *not* adding an encoding option to open, See:
    # https://github.com/pypa/virtualenv/issues/201#issuecomment-314569
    return codecs.open(os.path.join(here, *parts), 'r').read()

def find_version(*file_paths):
    version_file = read(*file_paths)
    version_match = re.search(r"^__version__ = ['\"]([^'\"]*)['\"]",
                              version_file, re.M)
    if version_match:
        return version_match.group(1)
    raise RuntimeError("Unable to find version string.")

long_description = read('README.rst')

tests_require = [
    'pytest',
    'mock',
    'pretend',
    'scripttest>=1.3',
    'virtualenv>=1.10',
    'freezegun',
]

setup(
    name="pip",
    version=find_version("pip", "__init__.py"),
    description="The PyPA recommended tool for installing Python packages",
    long_description=long_description,
    classifiers=[
        "Development Status :: 5 - Production/Stable",
        "Intended Audience :: Developers",
        "License :: OSI Approved :: MIT License",
        "Topic :: Software Development :: Build Tools",
        "Programming Language :: Python :: 2",
        "Programming Language :: Python :: 2.7",
        "Programming Language :: Python :: 3",
        "Programming Language :: Python :: 3.3",
        "Programming Language :: Python :: 3.4",
        "Programming Language :: Python :: 3.5",
        "Programming Language :: Python :: 3.6",
        "Programming Language :: Python :: Implementation :: PyPy"
    ],
    test_suite="tests",
    install_requires=tests_require + [
        'setuptools>=19.0',
    ],
)
```

What is subroutine/function

- In Math, we **use functions**, i.e. $\sin(x)$, \sqrt{x} , $f(x)$, ... which they have been **defined**.
- In subroutine, or Python's function, we **group up** a number of statements/computations and then **we give them a name**.
 - Enable to **reuse** those statements
 - Program will be shorter, easier to understand, less error-prone.
- A **function** is like a mini-program within a program.

Start our first function!!



```
def hello():  
    print('Hi there, CPE30!!')  
    print('This is my first function.')  
print("What's up")  
hello()  
hello()
```

- The first line is a **def statement** which defines a function named `hello()`.
- Function name follows the rules for creating an identifier.


Start our first function!!



```
def hello():  
    print('Hi there, CPE30!!')  
    print('This is my first function.')  
print("What's up")  
hello()  
hello()
```

- The code block after the **def statement** is the **body of the function**.
- This code is executed when the function is **called** not when the function is first defined.

Start our first function!!



```
def hello():  
    print('Hi there, CPE30!!')  
    print('This is my first function.')  
print("What's up")  
hello()  
hello()
```

- The `hello()` lines are **function calls**.
- When the program execution reaches these calls, it will jump to the top line in the function and begin executing the code there.
- When it reaches the end of function, the execution returns to the line that called and continues executing.

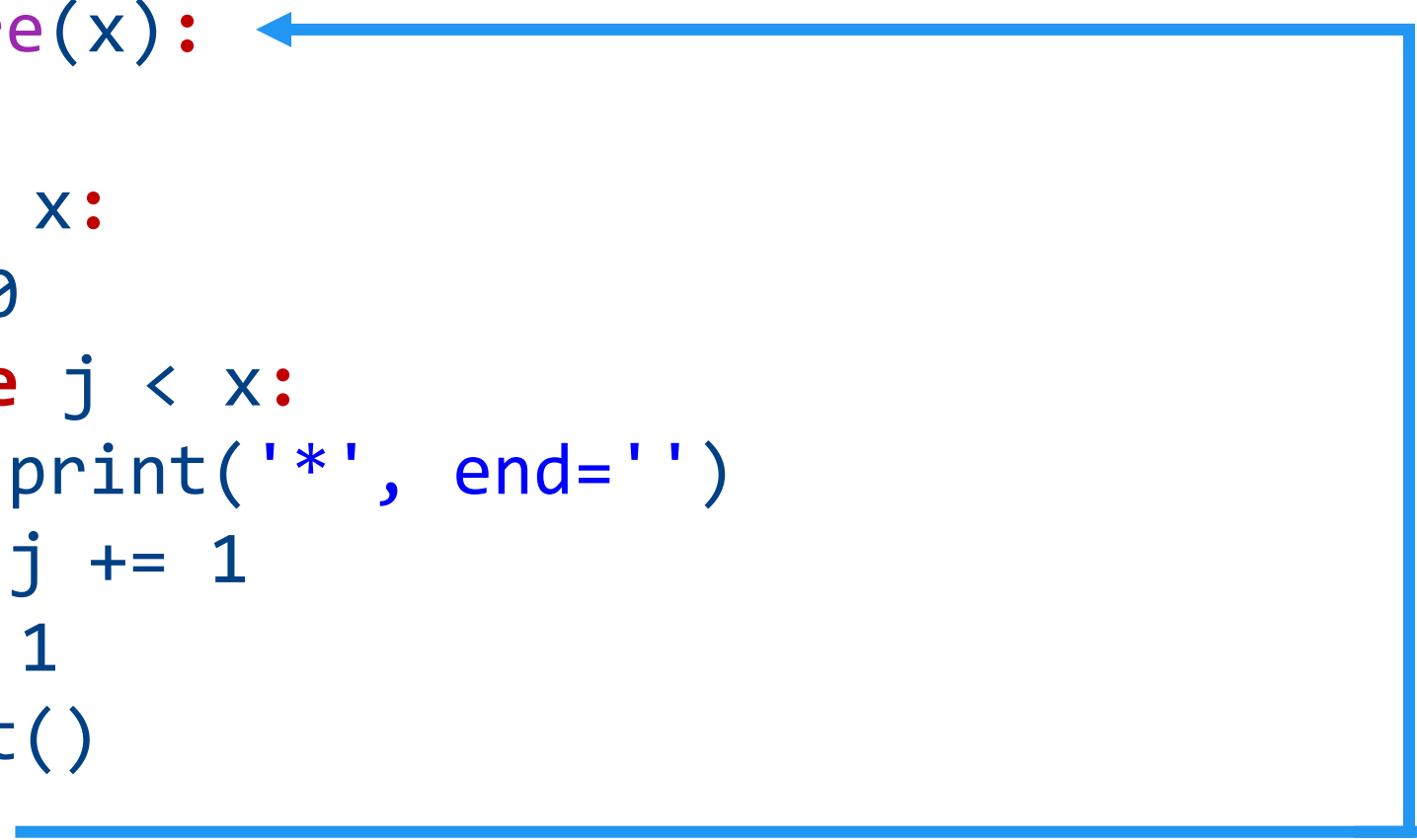
Passing Value

- When we call some function, we pass in values, called **arguments** or **parameters**, by typing them between the parentheses.

```
def functionName(arguments):  
    code block
```

Passing Value

```
01: def printSquare(x):  
02:     i = 0  
03:     while i < x:  
04:         j = 0  
05:         while j < x:  
06:             print('*', end=' ')  
07:             j += 1  
08:         i += 1  
09:         print()  
10: printSquare(5)
```

A blue arrow originates from the argument '5' in the function call 'printSquare(5)' on line 10 and points horizontally to the parameter 'x' in the function definition 'def printSquare(x):' on line 01. A vertical blue line descends from the end of the arrow on line 10, and another vertical blue line descends from the parameter 'x' on line 01, with a horizontal blue line connecting them at the right edge of the code block, forming a rectangular frame that highlights the argument passing mechanism.

Passing Value

```
01: def isADividedByB(a,b): ←
02:     if b == 0:
03:         print('error: b equal zero')
04:     elif a % b == 0:
05:         print('%i is divided by %i' %(a,b))
06:     else :
07:         print('%d is not divided by %d' %(a,b))
08: isADividedByB(50,5)
09: isADividedByB(33,6)
```


Task: *print triangle*

- Writing the printTriangle(n) which passing x as integer and printing the triangle.

INPUT:

Input size: 3

OUTPUT:

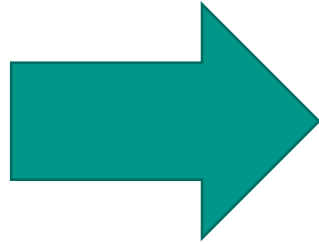
```
*  
***  
*****
```

Input size: 6

```
  *  
 ***  
*****  
*****  
*****  
*****  
*****
```

Task: *print triangle*

- **Hint:** If the problem cannot be solved as is, we rather decompose it into smaller parts and solve those smaller parts.
- Print N rows
 - Print spacebars
 - Print stars
 - Next row



```
def printTriangle(x) :  
    row = 0  
    while row < x :  
        printSpace(???)  
        printStar(???)  
        print()  
        row += 1
```

Task: *print triangle*

- Find the relationship of **row** and **space(dot)**

row	space	input = 7
0	6 *
1	5 * * *
2	4	. . . * * * * *
3	3	. . * * * * * *
4	2	. * * * * * * * *
5	1	. * * * * * * * *
6	0	* * * * * * * * *

$$\text{space} = \text{input} - \text{row} - 1$$

```
def printSpace(input,row) :  
    space = input - row -1  
    i = 0  
    while i < space :  
        print(' ',end='')  
        i += 1
```

Task: *print triangle*

- Find the relationship of **row** and **star**

row	star	input = 7
0	1 *
1	3 ***
2	5 *****
3	7 *****
4	9 *****
5	11 *****
6	13 *****

$$\text{star} = \text{row} * 2 + 1$$

```
def printStar(row) :  
    star = row * 2 + 1  
    i = 0  
    while i < star :  
        print('*',end='')  
        i += 1
```

Task: *print triangle*

```
def printSpace(input,row) :  
    space = input - row -1  
    i = 0  
    while i < space :  
        print(' ',end='')  
        i += 1  
  
def printStar(row) :  
    star = row * 2 + 1  
    i = 0  
    while i < star :  
        print('*',end='')  
        i += 1
```

```
def printTriangle(x) :  
    row = 0  
    while row < x :  
        printSpace(x,row)  
        printStar(row)  
        print()  
        row += 1  
  
a = int(input('Input size: '))  
printTriangle(a)
```

Returning Value

- The value to which a function call evaluates and send back to the caller is called the **return value** of the function.
- The **return statement** consists the **return** keyword and the **returning value** or **expression**.

```
def functionName(arguments):  
    code block  
    return returningValue
```

Task: *isPrime*

- Writing the `isPrime(x)` which passing x as integer and returning `True` when x is prime number and returning `False` when x is not.

Input integer: 70

70 is not a prime number.

Input integer: 23

23 is a prime number.

Task: *isPrime Ver.1*

```
01: a = int(input('Input integer: '))
02: i = 2
03: while i < a / 2:
04:     if a % i == 0:
05:         print('%d is not a prime number' %a)
06:         break
07:     i += 1
08: if i == a:
09:     print('%d is a prime number' %a)
```


Task: *isPrime Ver.2*

```
01: a = int(input('Input integer: '))
02: check = True
03: i = 2
04: while i < a / 2:
05:     if a % i == 0:
06:         check = False
07:         break
08:     i += 1
09: if check:
10:     print('%d is a prime number' %a)
11: else:
12:     print('%d is not a prime number' %a)
```

Task: *isPrime Ver.3*

```
01: def isPrime(x):
02:     check = True
03:     i = 2
04:     while i < x / 2:
05:         if x % i == 0:
06:             check = False
07:             break
08:         i += 1
09:     return check
10: a = int(input('Input integer: '))
11: check = isPrime(a)
12: if check:
13:     print('%d is a prime number' %a)
14: else:
15:     print('%d is not a prime number' %a)
```

Task: *isPrime Ver.4*

```
01: def isPrime(x):
02:     i = 2
03:     while i < x / 2:
04:         if x % i == 0:
05:             return False
06:         i += 1
07:     return True
08: a = int(input('Input integer: '))
09: if isPrime(a):
10:     print('%d is a prime number' %a)
11: else :
12:     print('%d is not a prime number' %a)
```

Returning Value

- For Python, we can return multiple value at the same time.

```
def plusMinus(a,b):  
    return a+b,a-b  
x=5;y=7  
plus,minus = plusMinus(x,y)  
print( '%i+%i=%i\n%i-%i=%i' ,%(x,y,plus,x,y,minus))
```

Task: *Triangle Area(Heron)*

- In geometry, **Heron's formula** gives the area of a triangle by requiring three sides of the triangle

$$area = \sqrt{s(s-a)(s-b)(s-c)}$$

; a, b, c are length of three sides of the triangle

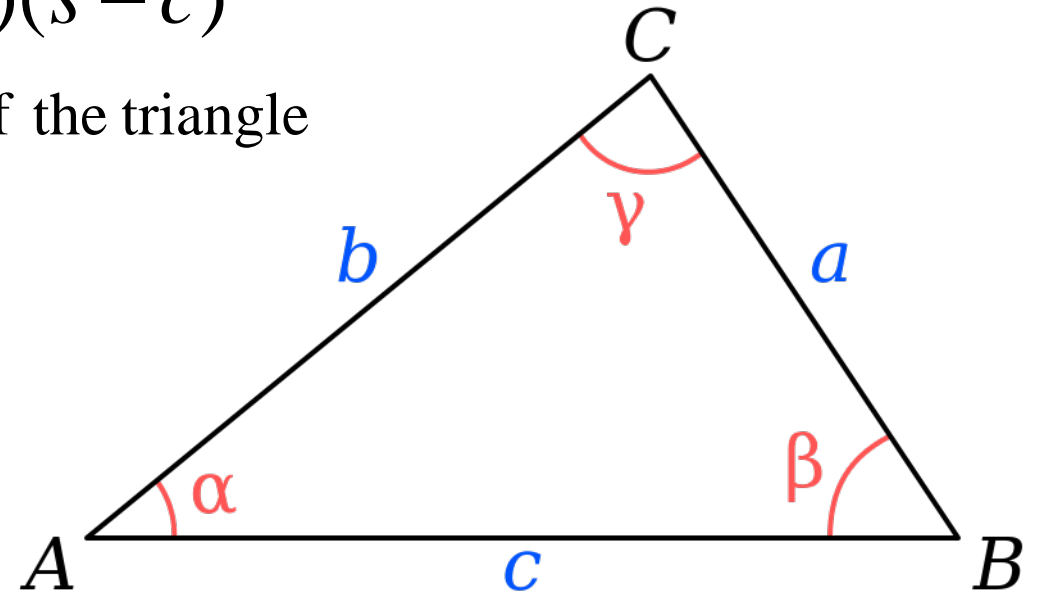
$$s = \frac{a + b + c}{2}$$

Input a: 6.8

Input b: 3.7

Input c: 4.6

Area: 8.02



Task: *Triangle Area(Heron)*

```
a = float(input('Input a: '))
b = float(input('Input b: '))
c = float(input('Input c: '))

s = (a+b+c)/2
area = (s*(s-a)*(s-b)*(s-c))**0.5
print('Area: %.2f' %area)
```

```
def getInput() :
    a = float(input('Input a: '))
    b = float(input('Input b: '))
    c = float(input('Input c: '))
    return a,b,c

def getArea(a,b,c) :
    s = (a+b+c)/2
    return (s*(s-a)*(s-b)*(s-c))**0.5

def printFormat(area) :
    print('Area: %.2f' %area)
    return

a,b,c = getInput()
area = getArea(a,b,c)
printFormat(area)
```

Task: *BMI and Weight Status*

- Write a program receive height(cm.) and weight(kg.) then calculate the **Body-Mass Index (BMI)** and show the **weight status**.

$$BMI = \frac{weight(kg)}{(height(m))^2}$$

```
Enter weight(kg.): 70  
Enter height(cm.): 2
```

```
BMI is 17.50, weight status: underweight
```

BMI	Weight Status
BMI < 18.5	Underweight
18.5 ≤ BMI < 25.0	Normal
25.0 ≤ BMI < 30.0	Overweight
BMI ≥ 30.0	Obese

Task: *BMI and Weight Status Ver.1*

```
01: weight = float(input('Enter weight(kg.): '))
02: height_cm = float(input('Enter height(cm.): '))
03: height_m = height_cm / 100
04: bmi = weight / (height_m**2)
05: status = ''
06: if bmi < 18.5 :
07:     status = 'underweight'
08: elif bmi < 25.0 :
09:     status = 'normal'
10: elif bmi < 30.0 :
11:     status = 'overweight'
12: else :
13:     status = 'obese'
14: print('BMI is %.2f, weight status: %s' %(bmi,status))
```


Task: *BMI and Weight Status Ver.2*

```
weight = float(input('Enter weight(kg.): '))
height_cm = float(input('Enter height(cm.): '))
height_m = cm_to_m_converter(height_cm)
bmi = bmi_calculator(weight, height_m)
status = get_bmi_status(bmi)
print('BMI is %.2f, weight status: %s' %(bmi,status))
```

Try writing these function by yourself:

```
cm_to_m_converter(cm)
bmi_calculator(w,m)
get_bmi_status(bmi)
```

Task: *BMI and Weight Status Ver.2*

```
def cm_to_m_converter(cm) :  
    return cm / 100  
def bmi_calculator(weight, height_m) :  
    return weight / (height_m**2)  
def get_bmi_status(bmi) :  
    if bmi < 18.5 :  
        return 'underweight'  
    elif bmi < 25.0 :  
        return 'normal'  
    elif bmi < 30.0 :  
        return 'overweight'  
    else :  
        return 'obese'
```

Task: *BMI and Weight Status Ver.3*

```
01: def get_bmi_with_status(height_cm,weight) :
02:     height_m = height_cm / 100
03:     bmi = weight / (height_m**2)
04:     if bmi < 18.5 :
05:         return bmi , 'underweight'
06:     elif bmi < 25.0 :
07:         return bmi , 'normal'
08:     elif bmi < 30.0 :
09:         return bmi , 'overweight'
10:     else :
11:         return bmi , 'obese'
12: weight = float(input('Enter weight(kg.): '))
13: height_cm = float(input('Enter height(cm.): '))
14: bmi , status = get_bmi_with_status(height_cm,weight)
15: print('BMI is %.2f, weight status: %s' %(bmi,status))
```

Conclusion Function Declaration

Function Definition Keyword

Function Name

```
def functionName(...) :  
    code block  
    return ...
```

Arguments / Parameters

ex.

() , (x) , (x,y) , (x,y,z) , ...

Return Statement

ex.

```
return  
return x  
return x + y  
return x , y  
return x , y , z
```