Lesson 5a: Subprograms

Functions

Modular Design

- A typical program can consist of thousands of lines of code
- A monolithic program is:
  - Unending
  - Incomprehensible
  - Difficult to:
    - Design
    - Write
    - Follow
    - Test
    - Re-use

```
OP_NOT_DEFINED = 1
DIV_ZERO = 2

operand1 = float(input("First operand: "))
operator = input("Operation: ")
operand2 = float(input("Second operand: "))

error = 0

if (operator == "+"):
    #Addition
    result = operand1 + operand2
elif (operator == "-"):
    #Subtraction
    result = operand1 - operand2
elif (operator == "*"):
    #Multiplication
    result = operand1 * operand2
elif (operator == "/"):
    #Division
    if (operand2 !=0):
        result = operand1 / operand2
    else:
        error = DIV_ZERO
else:
    #Operation not defined
    error = OP_NOT_DEFINED

if (not error):
    print(operand1,operator,operand2," = ",result)
else:
    if (error == OP_NO_DEFINIDA):
        print("ERROR: Operation not defined")
    elif (error == DIV_ZERO):
        print("ERROR: Division by zero")
```

What is the natural way we have
to solve complex problems?

Break the problem into smaller
subproblems that are **easier to solve**

Modular Design

- A modular design is based on the well-known strategy of "divide and conquer".

- Decompose the program into modules (independent parts)

- Each module
  - runs a single activity or task
  - is analyzed, designed and encoded separately

Modules are the bricks of computer scientists

Descending design

Abstraction:

WHAT DOES IT?

Description of the fundamental characteristics

Partition:

HOW IS IT DONE?

Definition of the parts that make it up.

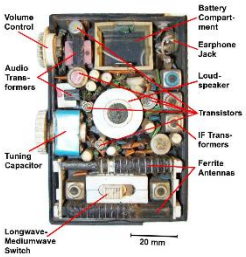
Radio

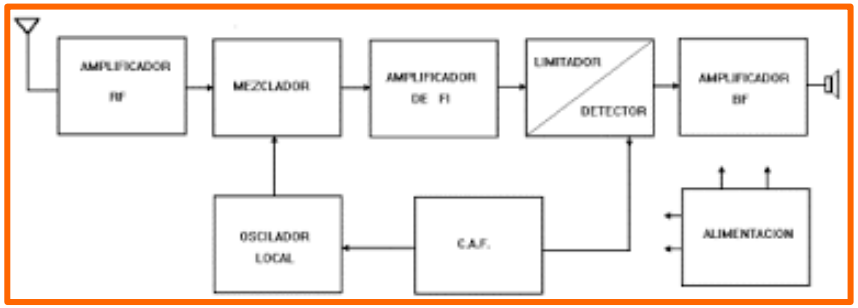
What does it?

Device that allows listening to sound signals, transmitted by a radio transmitter using electromagnetic waves.

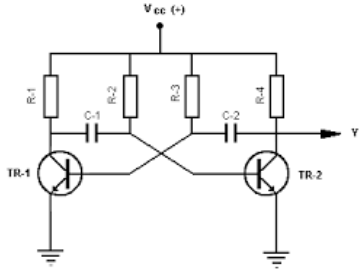
On/Off

Volume, Speakers, Tuner, Antenna





Oscillator

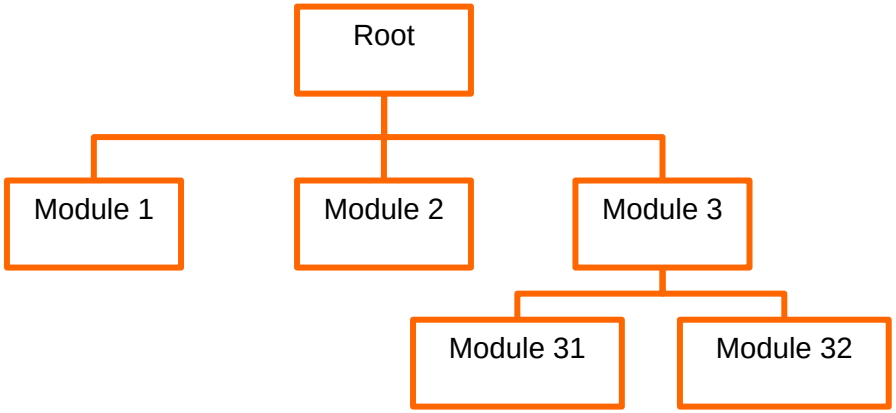


We apply these concepts to programming

- We are creating a structure through **decomposition**
  - Dividing a program into modules:
    - Self-sufficient
    - Breaking complexity
    - Reusable
    - Keep the code consistent and organized

- Avoid details with **abstraction**
  - Each module is a black box:
    - We cannot see the details
    - We do not need to see the details
    - We do not want to see the details
    - We hide the details

Descending design



Each program has a module called the **main program**.

If the **sub-module** is too *complex* it is **subdivided** into other sub-modules

Each module is encoded using the three basic control structures:

- Sequence
- Conditional
- Iterative

The **subdivision** process continues **until** the resulting sub-module performs only a **single task**

A module can **transfer control** to **sub-modules** in order to execute its function

When the sub-module finishes its task it returns the control to the module from which it received control

Python as a functional language

- Python implements different programming paradigms:
  - Functional
  - Object Oriented

- Depending on which one we choose, the modules will be different

- We will start by seeing the functional part of Python
  - Modules will be functions
  - Soon we will see that these modules can also be "objects"

Functions

- A function is a piece of code that can be reused every time it is needed.
- A function does not run unless it is called or invoked.
- A function has:
  - Unique name
  - Parameters (0 or more)
  - Body (instructions)
  - Return
  - Documentation (optional but recommended)

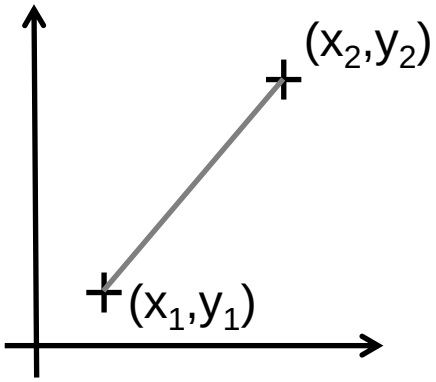
Advantages

- **Manageable:** It **reduces the problem** to a smaller, simpler and more understandable size.
- **Traceability**: Programs are **easier to modify**, and errors are easier to detect and correct.
- **Partition: Divisible programs** → Modules can be assigned to different teams or programmers:
  - Parallel work
  - Programming facilities: troubleshooting, testing, maintenance
- **Portability**: A module can be modified to work on another platform.
- **Reusable**: A module can be used in another project.

Attention: It is very important to define inputs, outputs and the task the module should do.

Library Functions

To make a program that calculates the distance between two points in space,...



...we need to read 4 real values (x_1, y_1, x_2, y_2) and then evaluate the formula:

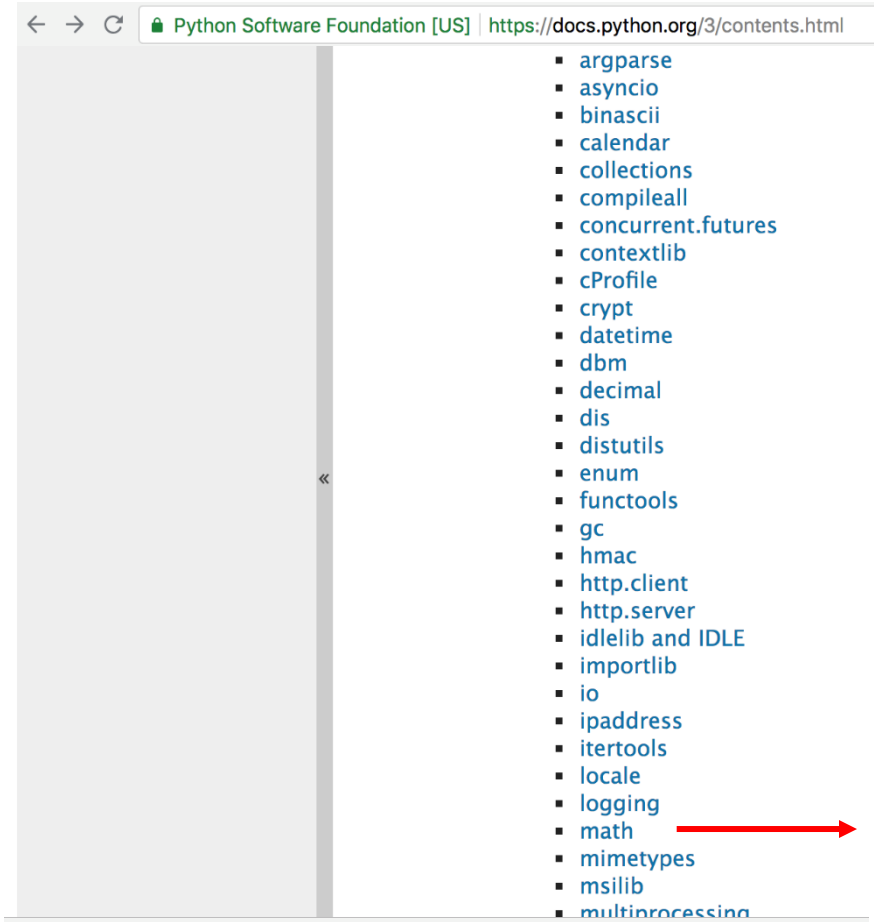
$$\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$$

Making a program that calculates the square root of a real number is complicated. It would take us a lot of time and effort to make sure the calculation was correct.

Many of the functions frequently used **have already been developed**.

Python libraries are basically this: functions already implemented and ready to be used, which do calculations that are frequently needed.

Library Functions



9.2.2. Power and logarithmic functions

`math.`**`exp`**(*x*)
Return *e* raised to the power *x*, where *e* = 2.718281... is the base of natural logarithms. This is usually more accurate than `math.e ** x` or `pow(math.e, x)`.

`math.`**`expm1`**(*x*)
Return *e* raised to the power *x*, minus 1. Here *e* is the base of natural logarithms. For small floats *x*, the subtraction in `exp(x) - 1` can result in a significant loss of precision; the `expm1()` function provides a way to compute this quantity to full precision:

```
>>> from math import exp, expm1
>>> exp(1e-5) - 1  # gives result accurate to 11 places
1.0000050000069649e-05
>>> expm1(1e-5)    # result accurate to full precision
1.0000050000166668e-05
```

New in version 3.2.

`math.`**`log`**(*x*[, *base*])
With one argument, return the natural logarithm of *x* (to base *e*).

With two arguments, return the logarithm of *x* to the given *base*, calculated as `log(x)/log(base)`.

`math.`**`log1p`**(*x*)
Return the natural logarithm of *1+x* (base *e*). The result is calculated in a way which is accurate for *x* near zero.

`math.`**`log2`**(*x*)
Return the base-2 logarithm of *x*. This is usually more accurate than `log(x, 2)`.

New in version 3.3.

See also: `int.bit_length()` returns the number of bits necessary to represent an integer in binary, excluding the sign and leading zeros.

`math.`**`log10`**(*x*)
Return the base-10 logarithm of *x*. This is usually more accurate than `log(x, 10)`.

`math.`**`pow`**(*x*, *y*)
Return `x` raised to the power `y`. Exceptional cases follow Annex 'F' of the C99 standard as far as possible. In particular, `pow(1.0, x)` and `pow(x, 0.0)` always return `1.0`, even when `x` is a zero or a NaN. If both `x` and `y` are finite, `x` is negative, and `y` is not an integer then `pow(x, y)` is undefined, and raises `ValueError`.

Unlike the built-in `**` operator, `math.pow()` converts both its arguments to type `float`. Use `**` or the built-in `pow()` function for computing exact integer powers.

`math.`**`sqrt`**(*x*)
Return the square root of *x*.

Example

```
## Program that computes the distance between 2 points

from math import sqrt, pow

x1 = float(input("Coordinate x first point :"))
y1 = float(input("Coordinate y first point :"))
x2 = float(input("Coordinate x second point :"))
y2 = float(input("Coordinate y second point :"))

dist = sqrt(pow(x1-x2,2)+pow(y1-y2,2))

print("The distance is",dist)
```

Example

```
## Program that computes the distance between 2 points

import math

x1 = float(input("Coordinate x first point :"))
y1 = float(input("Coordinate y first point :"))
x2 = float(input("Coordinate x second point :"))
y2 = float(input("Coordinate y second point :"))

dist = math.sqrt(math.pow(x1-x2,2)+math.pow(y1-y2,2))

print("The distance is",dist)
```

Example

We want to know how many different teams a basketball coach can do if there are 10 players in the squad.

$$\frac{n!}{r!(n-r)!} = \binom{n}{r}$$

combinations of n elements taken r at a time

n = 10 number of players in the squad
r = 5 number of players in the court

Let's write a program that can compute this for any sport (i.e. different number of players in the squad and different number of players in the court / field).

Example

```
n = int(input("Number of players on the squad: "))
r = int(input("Number of players in the court: "))
k = n-r

n_fact = 1
r_fact = 1
k_fact = 1

for n in range(n,1,-1):
      n_fact = n_fact * n

for r in range(r,1,-1):
    r_fact = r_fact * r

for k in range(k,1,-1):
    k_fact = k_fact * k

res=(n_fact)//(r_fact*k_fact)

print("The number of combinations is", res)
```

$$\frac{n!}{r!(n-r)!} = \binom{n}{r}$$

Functions

The three loops that calculate the factorial are the same. We have only changed `n` by `r` and then by `k`.

Each loop takes a value (`n`, `r` or `k`) and produces another (the factorials `n_fact`, `r_fact`, or `k_fact`) as output.

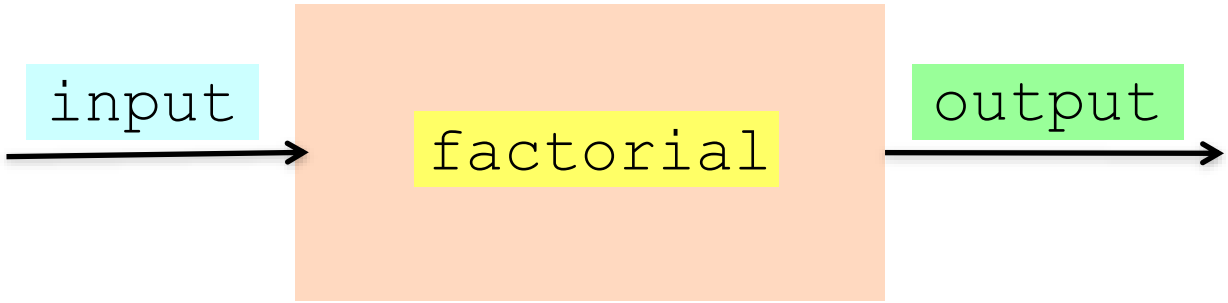
This suggests making a module that receives a value (int) and returns another (int).

We are applying **abstraction**. We know...

- what type of data we need as input (int),
- what the module does (calculates the factorial), and
- the output type (int).

So, we **do not need to know anything else to use it**.

How are functions declared?




```
def factorial(input):
    # Here: code to implement the function


    return output
```

`def`: Python reserved word

`Function Name`: Describe what the function does

`Parameters/Arguments`: Input data needed to perform calculations

`Body function`: instructions for performing calculations

`Return`: result obtained to be returned

How are functions declared?

To implement the function, we need to fill its body with the necessary instructions to do the task (in the example, calculating the factorial).

The body of the function must follow some rules:

1. Do not use `input()` to give values to function parameters.
2. Use the parameters as if they already have a value.

   In the example: it is necessary to think that the parameter num already has the value of which we want to calculate the factorial.
3. Do not use `print()` to return the result of the function.
4. To return the result of the function you must use the `return` command.

```
def factorial(num):
    fact = 1

    for n in range(num,1,-1):
        fact = fact*n
    return fact
```

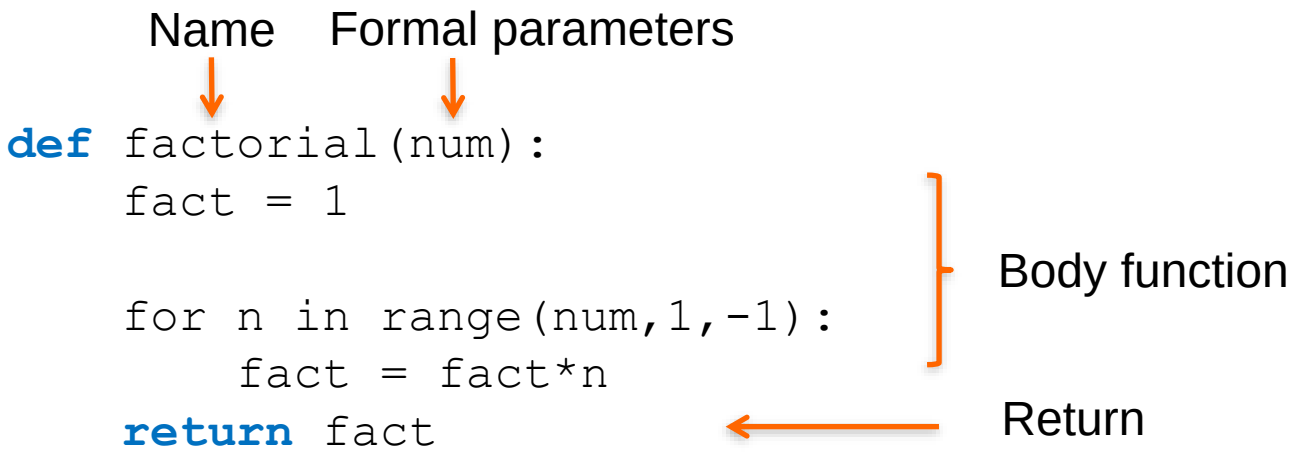
The variables used to implement the function → Are local variables of the function.

e.g. `fact` and `n` are local variables

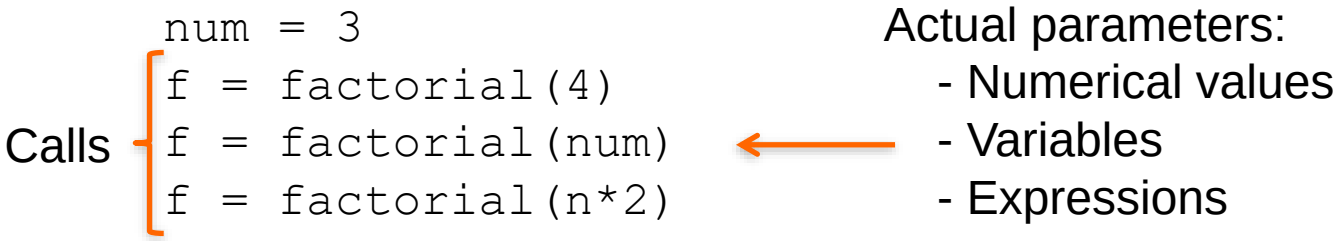
Local variables exist only while running the function where they are defined.

Definition vs. Call

Definition. It starts with the keyword **def** followed by the function name (in the example, `factorial`) and the formal parameters in parentheses (in the example, `num`)



Call. We invoke the function by its name followed by arguments (also called actual parameters) in parentheses.



Solution

```
def factorial(num):
    fact = 1
    for n in range(num,1,-1):
        fact = fact*n
    return fact



n = int(input("Number of players on the squad: "))
r = int(input("Number of players in the court: "))
k = n-r

n_fact = factorial(n)
r_fact = factorial(r)
k_fact = factorial(k)

res=(n_fact)//(r_fact*k_fact)

print("The number of combinations is", res)
```

Scope of a variable

When a function is called the formal parameters are assigned with the value of the actual parameters.

Each time a function is called, a new scope (a memory space) is created for the variables of the function.

In the following example, the formal and actual parameters have the same name but are different variables.

```
def factorial(num):
    fact = 1

    for n in range(num,1,-1):
        fact = fact*n
    return fact


num = 3
f = factorial(num)
```

Formal parameter

Function definition

Actual parameter

Main program:

- Initialization `num = 3`
- Call to function `factorial(num)`
- Assignment of result to variable `f`

How does the call work?

```
def factorial(num):
    fact = 1
    for n in range(num,1,-1):
        fact = fact*n
    return fact

n = int(input("Number of players on the squad: "))
r = int(input("Number of players in the court: "))
k = n-r


n_fact = factorial(n)
r_fact = factorial(r)
k_fact = factorial(k)

res=(n_fact)//(r_fact*k_fact)

print("The number of combinations is", res)
```

Scope of Main Program

factorial	code
n	10
r	5
k	5

Let's say we have entered these values

How does the call work?

```
def factorial(num):
    fact = 1
    for n in range(num,1,-1):
        fact = fact*n
    return fact

n = int(input("Number of players on the squad: "))
r = int(input("Number of players in the court: "))
k = n-r


n_fact = factorial(n)
r_fact = factorial(r)
k_fact = factorial(k)

res=(n_fact)//(r_fact*k_fact)

print("The number of combinations is", res)
```

Scope of Main Program

factorial	code
n	10
r	5
k	5

How does the call work?

```
def factorial(num):
    fact = 1
    for n in range(num,1,-1):
        fact = fact*n
    return fact

n = int(input("Number of players on the squad: "))
r = int(input("Number of players in the court: "))
k = n-r


n_fact = factorial(n)
r_fact = factorial(r)
k_fact = factorial(k)

res=(n_fact)//(r_fact*k_fact)

print("The number of combinations is", res)
```

Scope of factorial function

num	10

Scope of Main Program

factorial	code
n	10
r	5
k	5

How does the call work?

```
def factorial(num):
    fact = 1
    for n in range(num,1,-1):
        fact = fact*n
    return fact

n = int(input("Number of players on the squad: "))
r = int(input("Number of players in the court: "))
k = n-r


n_fact = factorial(n)
r_fact = factorial(r)
k_fact = factorial(k)

res=(n_fact)//(r_fact*k_fact)

print("The number of combinations is", res)
```

Scope of factorial function

num	10
fact	1

Scope of Main Program

factorial	code
n	10
r	5
k	5

How does the call work?

```
def factorial(num):
    fact = 1
    for n in range(num,1,-1):
        fact = fact*n
    return fact

n = int(input("Number of players on the squad: "))
r = int(input("Number of players in the court: "))
k = n-r


n_fact = factorial(n)
r_fact = factorial(r)
k_fact = factorial(k)

res=(n_fact)//(r_fact*k_fact)

print("The number of combinations is", res)
```

Scope of factorial function

num	10
fact	1
n	10

They have the same name but are different variables

Scope of Main Program

factorial	code
n	10
r	5
k	5

How does the call work?

```
def factorial(num):
    fact = 1
    for n in range(num,1,-1):
        fact = fact*n
    return fact

n = int(input("Number of players on the squad: "))
r = int(input("Number of players in the court: "))
k = n-r


n_fact = factorial(n)
r_fact = factorial(r)
k_fact = factorial(k)

res=(n_fact)//(r_fact*k_fact)

print("The number of combinations is", res)
```

Scope of factorial function

num	10
fact	3.628.800
n	2

Scope of Main Program

factorial	code
n	10
r	5
k	5

How does the call work?

```
def factorial(num):
    fact = 1
    for n in range(num,1,-1):
        fact = fact*n
    return fact

n = int(input("Number of players on the squad: "))
r = int(input("Number of players in the court: "))
k = n-r


n_fact = factorial(n)
r_fact = factorial(r)
k_fact = factorial(k)

res=(n_fact)//(r_fact*k_fact)

print("The number of combinations is", res)
```

Scope of factorial function

num	10
fact	3.628.800
n	2

Scope of Main Program

factorial	code
n	10
r	5
k	5

How does the call work?

```
def factorial(num):
    fact = 1
    for n in range(num,1,-1):
        fact = fact*n
    return fact

n = int(input("Number of players on the squad: "))
r = int(input("Number of players in the court: "))
k = n-r


n_fact = factorial(n)
r_fact = factorial(r)
k_fact = factorial(k)

res=(n_fact)//(r_fact*k_fact)

print("The number of combinations is", res)
```

Scope of Main Program

factorial	code
n	10
r	5
k	5
n_fact	3.628.800

Exercise: The clock

Write a program that reads the time of a clock from the keyboard and displays the time increased a second.

Times must be read and displayed in the following format: `05:23:52`
that is, hours, minutes and seconds with two digits and separated by a colon.

Write this program in **two steps**.

First version:

- Read the time in the specified format
- Get the hours, minutes and seconds
- Display the input time in the same format

Second version:

- Increase by 1 second controlling that if we go over the limits, we have to increase the next digit

Solution

```
# Input
time = input("Enter the time with format hh:mm:ss = ")

hh = int(time[0:2])
mm = int(time[3:5])
ss = int(time[6:8])

#Process
ss+=1

if (ss == 60):
    ss = 0
    mm += 1
if (mm == 60):
    mm =0
    hh += 1
if (hh == 24):
    hh = 0

#Output
new_time =""

# Format output when values < 10
if (hh<10):
    new_time += "0"
new_time += str(hh) + ":"
if (mm<10):
    new_time += "0"
new_time += str(mm) + ":"
if (ss<10):
    new_time += "0"
new_time += str(ss)

print(new_time)
```

Transform to two figures

Transform to two figures

Transform to two figures

Solution

```
def two_figures(num):
    out = ""
    if(num<10):
        out += "0”
    out += str(num)
    return out
```

```
# Input
time = input(" Enter the time with format hh:mm:ss = ")

hh = int(time[0:2])
mm = int(time[3:5])
ss = int(time[6:8])

#Process
ss+=1

if (ss == 60):
    ss = 0
    mm += 1
if (mm == 60):
    mm =0
    hh += 1
if (hh == 24):
    hh = 0

#Output
new_time =""

# Format output when values < 10
new_time += two_figures(hh) + ":"
new_time += two_figures(mm) + ":"
new_time += two_figures(ss)

print(new_time)
```

Function Call

Solution

```
def two_figures(num):
    out = ""
    if(num<10):
        out += "0”
    out += str(num)
    return out

# Input
time = input(" Enter the time with format hh:mm:ss = ")

hh = int(time[0:2])
mm = int(time[3:5])
ss = int(time[6:8])

#Process
ss+=1

if (ss == 60):
    ss = 0
    mm += 1
if (mm == 60):
    mm =0
    hh += 1
if (hh == 24):
    hh = 0

#Output
new_time =""

# Format output when values < 10
new_time += two_figures(hh) + ":"
new_time += two_figures(mm) + ":"
new_time += two_figures(ss)

print(new_time)
```

Modify the program to create a function that increases the time in 1 second.

The function will be named increment and will have as formal parameters the hours, minutes and seconds (in this order).

It will return the modified values.

Solution

```
def two_figures(num):
    out = ""
    if(num<10):
        out += "0”
    out += str(num)
    return out

def increment(hh,mm,ss):
    ss+=1

    if (ss == 60):
        ss = 0
        mm += 1
    if (mm == 60):
        mm =0
        hh += 1
    if (hh == 24):
       hh = 0
    return hh,mm,ss

# MAIN PROGRAM
# Input
time = input(" Enter the time with format hh:mm:ss = ")

hh = int(time[0:2])
mm = int(time[3:5])
ss = int(time[6:8])

#Process
hh,mm,ss = increment(hh,mm,ss)

#Output
new_time =""

# Format output when values < 10
new_time += two_figures(hh) + ":"
new_time += two_figures(mm) + ":"
new_time += two_figures(ss)

print(new_time)
```

Function Call

Exercise: Leap year?

Write a function that returns True (1) if leap year, otherwise, returns False (0)

To define the function, we must remember that we must establish:

- Function name
- Parameters
- Return

```
def is_leap_year(y):
    return ((y%4 == 0 and y%100 != 0) or y%400 == 0)
```

Exercise: days of month?

Perform a function that returns how many days a month has, based on the month and year that it receives as a parameter.

To define the function, we must remember that we must establish:

- Function name
- Parameters
- Return

```
def days_month(m,y):
    if (m==4 or m==6 or m==9 or m==11):
        ndays = 30
    elif (m==2):
        if (is_leap_year(y)):
            ndays = 29
        else:
            ndays = 28
    else:
        ndays = 31
    return ndays
```

Function Call

Exercise: Correct date?

Make a program using the functions above to check if an input date is correct.

Enter the date as dd/mm/yyyy (remember to separate day, month and year).

```
error = 0
data = input("Write a date in format dd/mm/yyyy: ")

dd = int(data[0:2])
mm = int(data[3:5])
yyyy = int(data[6:11])

if ((mm>=1) and (mm<=12)):
    ndays = days_month(mm,aaaa)
    if ((dd < 1) or (dd > ndays)):
        error=1
else:
    error=2

if (error == 0):
    print("Date is Correct")
elif(error == 1):
    print("Incorrect Day")
elif(error == 2):
    print("Incorrect Month")
```

Function Call