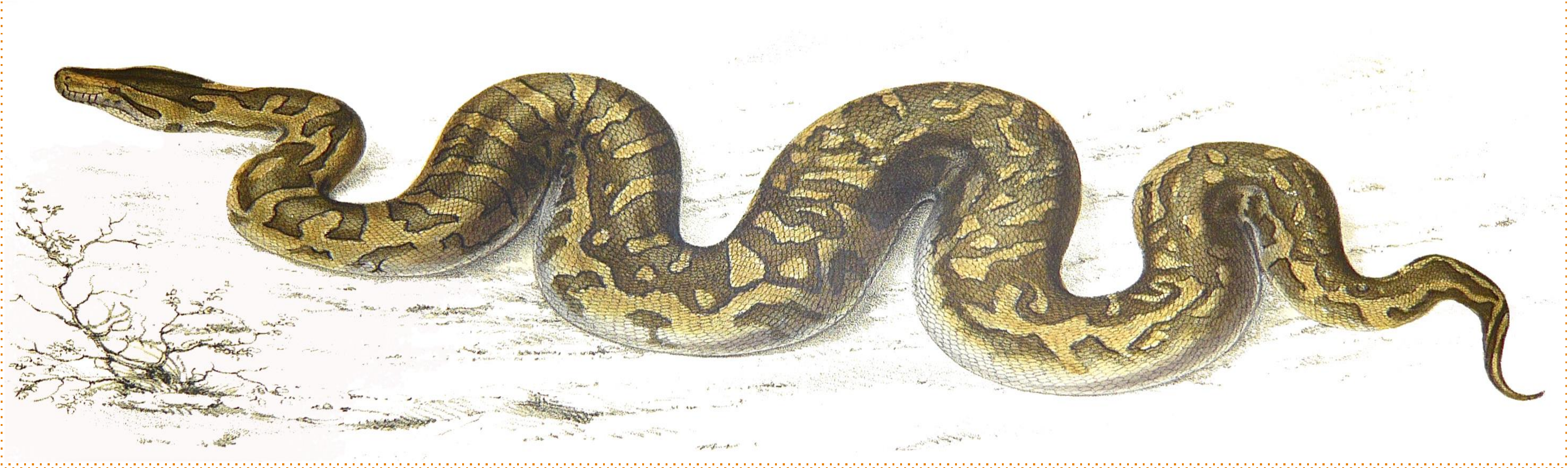


Python Programming

LECTURES BY

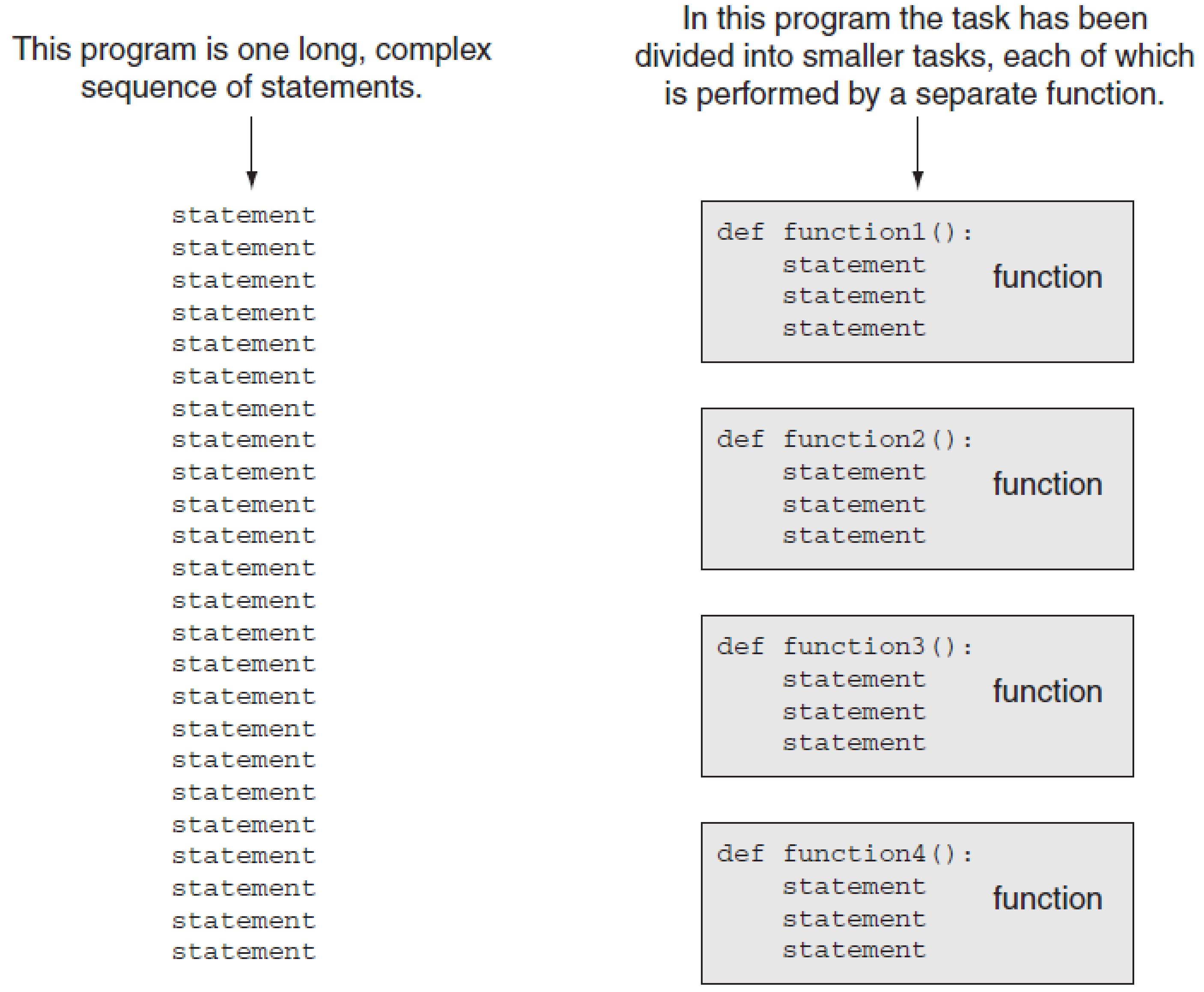
DR. AVINASH GULVE

Functions

- There are two kinds of functions in Python.
  - Built-in functions that are provided as part of Python –

    input(), type(), float(), int() ...
  - Functions that we define ourselves and then use
- We treat the of the built-in function names as "new" reserved words (i.e. we avoid them as variable names)

Using functions to divide and conquer a large task


This program is one long, complex sequence of statements.
In this program the task has been divided into smaller tasks, each of which is performed by a separate function.
function
function
function
function

```
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
```

```
def function1():
    statement
    statement
    statement
```

```
def function2():
    statement
    statement
    statement
```

```
def function3():
    statement
    statement
    statement
```

```
def function4():
    statement
    statement
    statement
```

Benefits of Modularizing a Program with Functions

The benefits of using functions include:

- Simpler code
- Code reuse
  - write the code once and call it multiple times
- Better testing and debugging
  - Can test and debug each function individually
- Faster development
- Easier facilitation of teamwork
  - Different team members can write different functions

Void Functions and Value-Returning Functions

A void function:

- Simply executes the statements it contains and then terminates.

A value-returning function:

- Executes the statements it contains, and then it returns a value back to the statement that called it.
  - The input, int, and float functions are examples of value-returning functions.

Defining and Calling a Function

Functions are given names

- Function naming rules:
  - Cannot use key words as a function name
  - Cannot contain spaces
  - First character must be a letter or underscore
  - All other characters must be a letter, number or underscore
  - Uppercase and lowercase characters are distinct

Function Definition

- In Python a function is some reusable code that takes arguments(s) as input does some computation and then returns a result or results
- We define a function using the **def** reserved word
- We call/invoke the function by using the function name, parenthesis and arguments in an expression
- Function name should be descriptive of the task carried out by the function
- Function definition: specifies what function does

```
def function_name(arguments):
        statement
        statement
```

Defining and Calling a Function (cont'd.)

- Function header: first line of function
  - Includes keyword def and function name, followed by parentheses and colon
- Block: set of statements that belong together as a group
  - Example: the statements included in a function
- Call a function to execute it
  - When a function is called:
    - Interpreter jumps to the function and executes statements in the block
    - Interpreter jumps back to part of program that called the function- Known as function return

Indentation in Python

- Each block must be indented
  - Lines in block must begin with the same number of spaces
    - Use tabs or spaces to indent lines in a block, but not both as this can confuse the Python interpreter
    - IDLE automatically indents the lines in a block
  - Blank lines that appear in a block are ignored

Defining Functions

Function definition begins with “def.”

Function name and its arguments.

```
def get_final_answer(filename):
    “““Documentation String”””
    line1
    line2
    return total_counter
```

Colon.

The indentation matters…
First line with less
indentation is considered to be
outside of the function definition.

The keyword ‘return’ indicates the
value to be sent back to the caller.

```
def max (a, b):
  “’return maximum among a and b’’
  if (a > b):
    return a
  else:
    return b
```

keyword

Function Name

2 arguments
a and b
(formal args)

```
x = max(6, 4)
```

Body of thefunction, indented w.r.t the def keyword

Call to the function.
Actual args are 6 and 4.

Documentation comment (**docstring**), type
help <function-name>
on prompt to get help for the function

```
>>>def max (a, b):
        '''return maximum among a and b'''
        if (a > b):
            return a
        else:
            return b
```

```
>>>help(max)
Help on function max in module __main__:

max(a, b)
    return maximum among a and b
```

Function Definition: Syntax

Syntax:

```
def functionname( parameters ):
   "function_docstring"
   function_body
   return [expression]
```

By default, parameters have a positional behavior, and you need to inform them in the same order that they were defined.

Example:

```
def printme( str ):
   "This function prints a passed string"
   print str
   return
```

Printing docstring

```
def printme( str ):
        "This function prints a passed string"
        print(str)
        return
print(printme.__doc__)

o/p
This function prints a passed string
```

Use of docstring for help

```
>>> def printme( str ):
      "This function prints a passed string"
      print(str)
      return

>>> help(printme)
Help on function printme in module __main__:

printme(str)
    This function prints a passed string
```

Calling a function

The syntax for a function call is:

```
>>> def myfun(x, y):
        return x * y
>>> myfun(3, 4)
12
```

Parameters in Python are *Call by Assignment*

- Old values for the variables that are parameter names are hidden, and these variables are simply made to *refer to* the new values
- All assignment in Python, including binding function parameters, uses *reference semantics.*

Functions without return

All functions in Python have a return value, even if no *return* line inside the code

Functions without a *return,* return the special value *None.*

- *None* is a special constant in the language
- *None* is used like *NULL*, *void*, or *nil* in other languages
- *None* is also logically equivalent to False
- The interpreter’s REPL doesn’t print *None*

Function Arguments

A function can be categorized by using the following types of formal arguments::

1. Required arguments
2. Keyword arguments
3. Default arguments
4. Variable-length arguments

1. Required arguments

Required arguments are the arguments passed to a function in correct positional order.

```
def addab(a,b):
    print(“a= %4d and b= %4d”%(a,b))
    sum=a+b
    return(sum)

p=int(input("Enter a number: "))
q=int(input("Enter another number: "))

res=addab(p,q)
print("Sum is ",res)
```

2. Keyword arguments

- Can call a function with some/all of its arguments out of order as long as you specify their names
- Can be combined with default arguments.

```
def addabc(a,b,c):
   print("a=%4d,b=%4d,c=%4d"%(a,b,c))
   sum=a+b+c
   return(sum)
p=int(input("Enter first number: "))
q=int(input("Enter second number: "))
r=int(input("Enter third number: "))

res=addabc(b=q,a=p,c=r)
print("Sum is ",res)

res=addabc(p,c=r,b=q)
print("Sum is ",res)
```

```
Enter first number: 10
Enter second number: 20
Enter third number: 30
a=  10,b=  20,c=  30
Sum is  60
a=  10,b=  20,c=  30
Sum is  60
```

3. Default Arguments

- You can provide default values for a function’s arguments
- A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument.
- These arguments are optional when the function is called

```
def addabc(a=10,b=20,c=30):
   print("a=%4d,b=%4d,c=%4d"%(a,b,c))
   sum=a+b+c
   return(sum)

p=int(input("Enter first number: "))
q=int(input("Enter second number: "))
r=int(input("Enter third number: "))

res=addabc(p,q)
print("Sum is ",res)

res=addabc(p,q,r)
print("Sum is ",res)
```

```
Enter first number: 1
Enter second number: 2
Enter third number: 3
a=   1,b=   2,c=  30
Sum is  33
a=   1,b=   2,c=   3
Sum is  6
```

4. Variable-length arguments

- You may need to process a function for more arguments than you specified while defining the function. These arguments are called *variable-length* arguments and are not named in the function definition, unlike required and default arguments.

```
def add(*args):
    sum=0
    for n in args:
        sum = sum + n
    return sum

res=add(10,20,30)
print("Sum = ",res)

res=add(10,20,30,40,50)
print("Sum = ",res)
```

Nested functions

A function which is defined inside another function is known as nested function and is sometimes called an "inner function".

Nested functions are able to access variables of the enclosing scope.

In **Python**, these non-local variables are read-only by default and we must declare them explicitly as non-local (using nonlocal keyword) in order to modify them

Nested functions

```
def add(a):
    def sum(b):
        return(a+b)
    return(sum(20))

print(add(5))
```

```
md=multi_digits(sum)
return(md)
```

```
def add(a,b):
    sum=a+b
    def multi_digits(n):
        m=1
        while(n>0):
            m=m*(n%10)
            n=n//10
        return(m)
    print("Sum = ",sum)
    return(multi_digits(sum))
print(add(25,72))
```

Scope of variable in nested functions

```
def first():
    a=["Government"]     #Iterable
    def second():
        a[0]="college"
        print("Value of a from second()",a)
    second()
    print("Value of a from first()",a)

first()
```

Scope of variable in nested functions

```
def first():
    a=10
    def second():
        nonlocal a
        a=20
        print("Value of 'a' from second()",a)
    second()
    print("Value of 'a' from first()",a)

first()
```

```
def first():
    first.a=10
    def second():
        first.a=20
        print("Value of 'a' from second()",first.a)
    second()
    print("Value of 'a' from first()",first.a)

first()
```

Nested functions returning function

```
def first(a):

    def second():
        print("Value of variable 'a' is ",a)

    return second

f=first(10)
f()
```

Nested functions returning function

```
def first(a,b):
    sum=a+b
    def second():
        n=sum
        m=1
        while(n>0):
            m=m*(n%10)
            n=n//10
        print("Multiplication result is ",m)
    print("sum = ",sum)
    return second

f=first(10,15)
f()
```

Function as Argument

```
def add_ab(a,b):
    sum=a+b
    return(sum)

def read_2_num(func):
    p=int(input("Enter first number: "))
    q=int(input("Enter first number: "))
    res=func(p,q)
    print("sum of 'p' and 'q' is ",res)

read_2_num(add_ab)
```

Local Variables

- Local variable: variable that is assigned a value inside a function
  - Belongs to the function in which it was created
    - Only statements inside that function can access it, error will occur if another function tries to access the variable
- Scope: the part of a program in which a variable may be accessed
  - For local variable: function in which created
- Local variable cannot be accessed by statements inside its function which precede its creation
- Different functions may have local variables with the same name
  - Each function does not see the other function’s local variables, so no confusion

Global Variables

Global variable: created by assignment statement written outside all the functions

- Can be accessed by any statement in the program file, including from within a function
- If a function needs to assign a value to the global variable, the global variable must be redeclared within the function
  - General format: global *variable_name*

Global Variable - example

```
a=10
def modify_a():
    global a #if want to change value of a then define it as global
    print("Value of global variable a before modification",a)
    a=20
    print("Value of global variable a after modification",a)

print("Value of global variable before invocation of function",a)
modify_a()
print("Value of global variable after invocation of function",a)
```

Command Line Arguments

- The Python sys module provides access to any command-line arguments via the sys.argv. This serves two purpose:
  - sys.argv is the list of command-line arguments.
  - len(sys.argv) is the number of command-line arguments.
- Here sys.argv[0] is the program ie. script name.

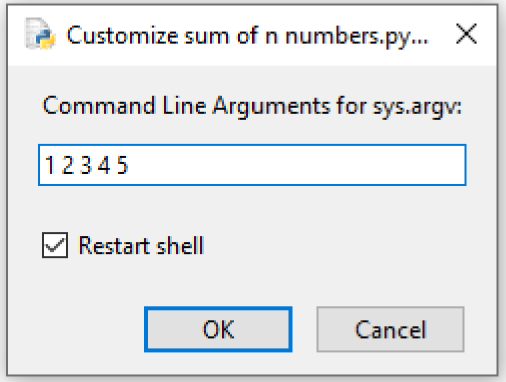
Executing the program

- on the dos prompt, give the command

```
c:\cmd>python sum.py 23 45 67 89 124
```

Or

From IDLE- select Run and Run..customized



Command Line Arguments for sys.argv:

```
1 2 3 4 5
```

[✓] Restart shell

Command Line Arguments

```
import sys
n=len(sys.argv)
print("No of arguments passed: ",n)
sum=0
for i in range(1,n):
    sum = sum+int(sys.argv[i])
print("Sum = ",sum)
```

Function returning multiple values

- Function can return multiple values using return statement. The values are separated by comma. e.g.

```
return a,b,c,d
```

- In Python, comma-separated values are considered tuples without parentheses. For this reason, the function in the above example returns a tuple with each value as an element.

Returning multiple values

```
def swap(a,b):
    a,b=b,a
    return(a,b)

p=int(input("Enter first number: "))
q=int(input("Enter second number: "))
print("Before swapping")
print("p= %4d, q= %4d"%(p,q))
p,q=swap(p,q)
print("After swapping")
print("p= %4d, q= %4d"%(p,q))
```

Using tuple

```
def swap(a,b):
    a,b=b,a
    return(a,b)

p=int(input("Enter first number: "))
q=int(input("Enter second number: "))
print("Before swapping")
print("p= %4d, q= %4d"%(p,q))
t=()
t=swap(p,q)
print("After swapping")
print("p= %4d, q= %4d"%(t[0],t[1]))
```

Using List

```
def swap(a,b):
    a,b=b,a
    return(a,b)

p=int(input("Enter first number: "))
q=int(input("Enter second number: "))
print("Before swapping")
print("p= %4d, q= %4d"%(p,q))
lst=[]
lst=swap(p,q)
print("After swapping")
print("p= %4d, q= %4d"%(lst[0],lst[1]))
```

Recursion

Consider 5! = 5x4x3x2x1

Can be re-written as 5!=5x4!

In general n! = nx(n-1)!

factorial(n) = n * factorial(n-1)

Recursion

- No computation in first phase, only function calls
- Deferred/Postponed computation –after function calls terminate, computation starts
- Sequence of calls have to be remembered

```
Fact(4)
4 * fact(3)
4 * (3 * fact(2))
4 * (3 * (2 * fact(1)))
4 * (3 * (2 * 1))
4 * (3 * 2)
4*  (6)
24
```

Recursion

- Size of the problem reduces at each step
- The nature of the problem remains the same
- There must be at least one terminating condition
- Easy from programing point of view
- May not efficient computation point of view

Recursion example

```
def power(b,n):
    if n==1:
        return 1
    else:
        return b*power(b,n-1)

no,expo =input("Enter the number and exponent: ").split()
res= power(int(no),int(expo)+1)
print("Result = ",res)
```

Modules

- As program gets longer, need to organize them for easier access and easier maintenance.
- Reuse same functions across programs without copying its definition into each program.
- Python allows putting definitions in a file
  - Use them in a script or in an interactive instance of the interpreter
- Such a file is called a *module*
  - Definitions from a module can be *imported* into other modules or into the *main* module

Modules

- A module’s definitions can be imported into other modules by using “import *name*”
- The module’s name is available as a global variable value
- To access a module’s functions, type *“name.function()”*
- Modules can contain executable statements along with function definitions
- Modules can import other modules
- Can import names from a module into the importing module’s symbol table

  from mod import m1, m2 (or **)*

Importing modules

- A file containing a set of functions you want to include in your application.
- To create a module just save the code you want in a file with the file extension .py:
- Use the module just created, by using the import statement
- You can create an alias when you import a module, by using the as keyword:

Importing module- example

```
#add_ab.py
def addab(a,b):
   return(a+b)
```

$\longrightarrow$ Name of the module add_ab.py

```
#res.py
from add_ab import *
a=10
b=20
res=addab(a,b)
print(res)
```

$\longrightarrow$ Module add_ab imported in res.py

Importing module- example

```
#add_ab.py
def addab(a,b):
   return(a+b)
```

Name of the module add_ab.py

```
import add_ab
a=10
b=20
add=add_ab.addab
res=add(a,b)
print(res)
```

New name given to the function

```
#res1.py
import add_ab
a=10
b=20
res=add_ab.addab(a,b)
print(res)
```

Importing Module add_ab

```
#res2.py
import add_ab as ad
a=10
b=20
res=ad.addab(a,b)
print(res)
```

New name given to the import module

__main__ in Modules

By adding this code at the end of your module

```
if __name__ == "__main__":
    ... # Some code here
```

you can make the file usable as a script as well as an importable module

```
#add_ab.py
def addab(a,b):
    return(a+b)

#if __name__ == "__main__":
res=addab(10,20)
print("Sum = ",res)
```

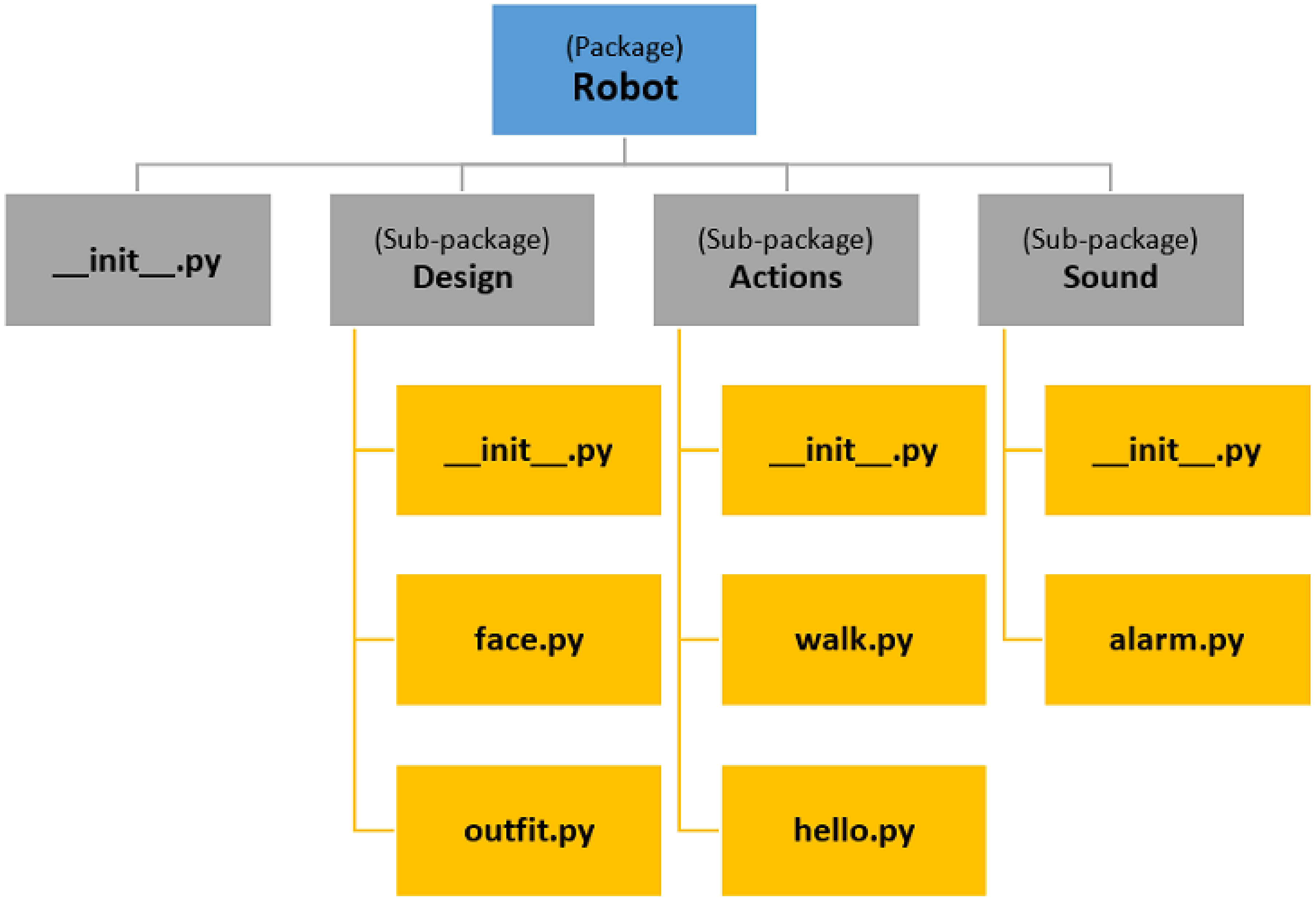
```
#result.py
import add_ab
a=15
b=28
res=add_ab.addab(a,b)
print(res)
```

Package

- A Python package is a collection of Python modules.
- Another level of *organization.*
- *Packages* are a way of structuring Python's module namespace by using *dotted module names*.
  - The module name A.B designates a sub-module named B in a package named A.
  - The use of dotted module names saves the authors of multi-module packages like NumPy or Pillow from having to worry about each other's module names.
- To tell Python that a particular directory is a package, we create a file named __init__.py inside it and then it is considered as a package

__init.py__

- The __init__.py files are required to make Python treat directories containing the file as packages.
- This prevents directories with a common name, such as string, unintentionally hiding valid modules that occur later on the module search path.
- __init__.py can just be an empty file
- It can also execute initialization code for the package


(Package)
Robot
__init__.py
(Sub-package)
Design
(Sub-package)
Actions
(Sub-package)
Sound
__init__.py
face.py
outfit.py
__init__.py
walk.py
hello.py
__init__.py
alarm.py

How to create a package

- First, we create a directory and give it a package name, preferably related to its operation.
- Then we put the classes and the required functions in it.
- Finally we create an __init__.py file inside the directory, to let Python know that the directory is a package.