Functions – and Program Structuring

INDRAPRASTHA INSTITUTE *of*
INFORMATION TECHNOLOGY
DELHI

Recap

- We have seen program as a monolith sequence of statements
- Main types of statements
  - Assignment - var = expression
  - Conditional - if-then-else, if-then, if-elifs
  - Iteration - for loop, while loop
- These statements are sufficient to compute anything computable
- For a large program, or a complex problem
  - One monolith seq of statements is hard to construct or debug
  - Having only the above statements makes it harder
- Functions provide an answer to both of these
  - Allows us to build new and more powerful "constructs" from the basic language constructs, which we can use in our code
  - Allows code to be broken into pieces

Functions

- In math, we have functions like:

  $Z = f(x,y)$

- After defining a function, we can use it in other functions
- In python, we can define very general functions, and use them
- Like in math, functions may have parameters, and to compute a function, values of parameters have to be provided
- Function is a unit of computation – which can be invoked from different places, i.e. used wherever we want
- With functions, a python program is a set of function declarations, and a “main program” which calls / uses these functions
- Let's show it by example

Python Functions: Example

```
# defining a fn sq
def square(x):
    return x*x

# defining a fn cube
def cube(y):
    return y*y*y

# Main program
a, b = 2, 4
c = square(a) + cube(b)
print("Val of c: ", c)
```

- Two functions defined - each has one parameter
- Code of function definition specifies the computation the fn does
- Function can return some value
- To use the function – it is called, value of parameter is provided
- On call – parameter gets value, body of function executed; value returned (and can be used)

Defining a function

- We need two basic capabilities - defining a function and calling a defined function
- Defining a function is done by def

```
def fn_name(parm-list):
    <fn-body>
```

- Parameters are optional; parameters are available for use in body
- The function execution terminates when it executes a return statement, or its body completes

Calling a function

- Defining a function just defines it, to execute we must call it
- Statement to call a function: just the function name with parameters:

```
fn_name(arguments)
```

- If function does not have any arguments, it must be called with () - this tells the interpreter that this is not a variable but a function call

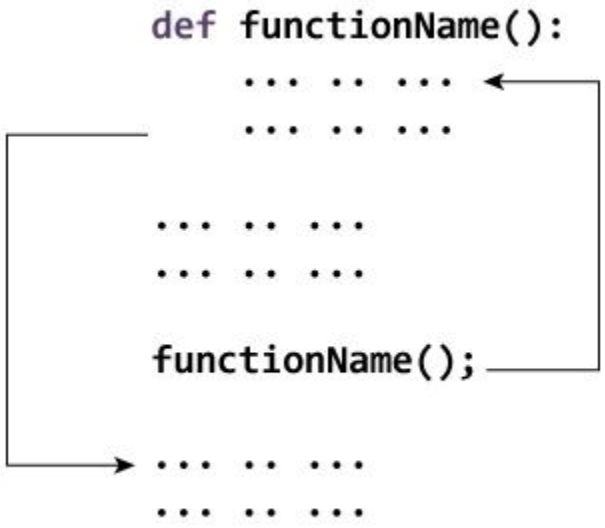
```
fn_name()
```

- If function has parameters, arguments need to be provided for all the parameters - provide value of the parameter for fn execution

Defining and Calling: flow of control

- Program is a sequence of statements being executed by interpreter
- Function definition is a definition - not an executable statement
- Function can be defined anywhere in the sequence of statements, though good practice is to define them at the start
- Function call is an executable statement
- On encountering a function call statement, to execute it:
  - Interpreter jumps to function definition
  - Parameters are assigned values that corresponding arguments in the calling statement have
  - Body of the function (a sequence of statements) is executed
  - Upon completion of the function, the control returns to the calling stmt
  - Return value, if any, is used where the function was called

Flow of control – diagram (programiz)



Executing a Program with Functions

A general program structure:

```
def fn1():
    body
def fn2():
    body
def fn3(params):
    body
# Main program
Stmt-block
# includes some call stmts
```

When interpreter gets this program

- On function definitions, it records some information; body not executed
- Starts execution from the first executable stmt in the stmt block of the main program
- On a call statement, control is transferred to the function; function starts executing
- On return statement in the function, goes back to the call stmt (in the main program)
- Execution continues in the main program
- Note: Function definition must be before the function call stmt is executed. Otherwise results in error.

Return statement

- A function can use in its body a special statement:

```
return <expression> # expression is optional
```

- Return statement serves two purposes
  - Terminates the execution of the function and returns the control back to where the function was called
  - Returns a value to the caller
- A function execution can also terminate when its body finishes
  - Like having a return statement as the last statement
- If some value specified in return - that is provided at calling point
- Otherwise the return value is treated as `None` (a special value)

Note: In Python, functions can return multiple values. Just write each value/expression after return, separated by commas.

Visualize execution using pythontutor

- We can visualize the execution of a program using pythontutor.com
- Shows the main program execution in global frame, which has the variables and their objects
- When a function is called it creates a new frame for the function - executes the function code in that frame
- When the function returns, the function frame is deleted and the control returns to global frame with the return value
- Pythontutor.com - is free for anyone to use: a good tool to use; you can also write simple programs directly in it

Example

What will be the output of the following program?

```
def add(a,b):
    return a+b
    print("hello", end="")


sum=add(1,2)
print(sum)
```

Note: setting end parameter to "" in the print function will print the values without any space or new line.

eg:

```
print(3, end="")
print(5)
```

```
Output: 35
```

Solution: 3

print statement inside the add function will not be executed because it is after the return statement.

Quiz – Single Correct

Order in which names of colors are printed when the program is executed?

A. Red, Yellow, Blue, Green
B. Red, Green, Yellow, Blue
C. Yellow, Green, Blue, Red
D. Red, Yellow, Green, Blue

```
print("Red")
def f(a, b):
    s = a*b
    print("Green")
    return s

print("Yellow")
num1 = 10
num2 = 5
ans = f(10, 5)
print("Blue")
```

Quiz – Single Correct

Order in which names of colors are printed when the program is executed?

A. Red, Yellow, Blue, Green
B. Red, Green, Yellow, Blue
C. Yellow, Green, Blue, Red
D. Red, Yellow, Green, Blue

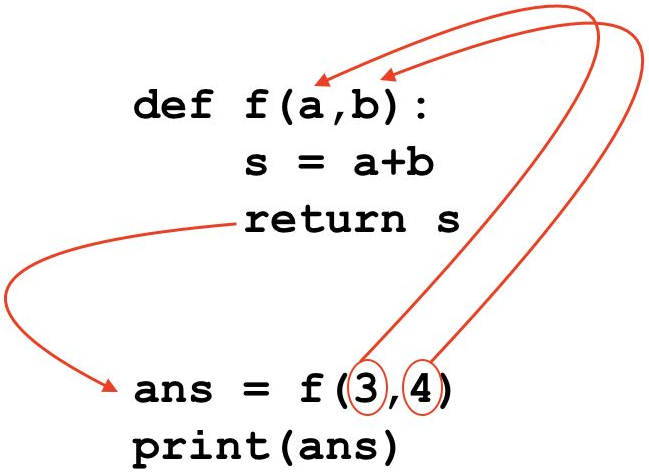
```
print("Red")#1
def f(a, b): #6
    s = a*b #7
    print("Green") #8
    return s #9

print("Yellow")#2
num1 = 10 #3
num2 = 5 #4
ans = f(10, 5) #5
print("Blue") #10
```

Argument Passing

- A function definition may have ***parameters*** (or not) - these are available inside the function for use
- Call to a function has ***arguments*** (or not)
- When a function is called, argument values are assigned to the parameters
- Positional arguments (also called required arguments) - arguments are assigned to parameters in order
  - Must have same number of arguments for calling
  - i-th argument value is assigned to i-th parameter
- When a function is called, interpreter checks if the # of args is same as # of parms
  - If number of arguments is not same, error

Argument passing example



- The values 3 and 4 are passed as arguments for function `f`.
- The arguments values are copied to function parameters `a` and `b`.
- `a` and `b` are used for computing value of `s`.
- `s` is returned by the function and assigned to variable ans.
- The variable ans now holds the value 7 and is printed.

Argument Passing ...

- Arguments can be pass by value or pass by reference
  - Pass by value - the value of arg is assigned to the parm
  - Pass by reference - a reference to the arg is assigned to the parm - in this case changes made by function can be reflected in the caller
  - Python uses pass by value, but in some cases, this value is a reference - we will discuss it later
- Complex objects can also be passed as arguments

Parameters vs Arguments

Parameter: Parameter is a variable used during the function definition inside the parentheses after the function name.

Argument: An argument is the actual value passed to the function while making a function call.

```
def add(a,b):
    return a+b

#Here a and b are the parameters

sum= add(1,3)
#Here 1 and 3 are the arguments
```

Quiz

Which of the following demonstrates valid usage of function?

Program to print the sum of two numbers a and b.

a)
```
def add(a,b):
    return a+b
a,b = 5,7
sum=add()
print(sum)
```

b)
```
a,b = 5,7

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

c)
```
def add(a,b):
    return a+b
a,b = 5,7
sum=add(a,b)
print(sum)
```

d)
```
a,b = 5,7
sum=add(a,b)
print(sum)

def add(a,b):
    return a+b
```

Quiz(Solution)

Which of the following demonstrates valid usage of function?

Program to print the sum of two numbers a and b.

A) ❌
B) ✅
C) ✅
D) ❌

a)
```
def add(a,b):
    return a+b
a,b = 5,7
sum=add()
print(sum)
```

b)
```
a,b= 5,7

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

c)
```
def add(a,b):
    return a+b
a,b= 5,7
sum=add(a,b)
print(sum)
```

d)
```
a,b = 5,7
sum=add(a,b)
print(sum)

def add(a,b):
    return a+b
```

Local Variables

- For now we have: functions and main program
  - Fns are defined; main program is the sequence of statements at the top level
- All vars defined in the function are local - they exist only when function is executing
- Parameters of a function are also local variables
- Local variables of a function can only be accessed from within the function
- Main program cannot access local vars of functions
- As vars in a function are local to a function - many fns can have same var names with no conflict
- Examples - main accessing local, two fns having same var

Global Variables

- Variables of main can be accessed within a function - these are called global variables, which can be accessed in any function
- If a variable x is used in a function, the interpreter first looks for x in the function frame (i.e. is it in the function)
- If not, it will look for x in the main program - if it exists there, the value of x as defined in main will be used
- If not in main, then an error

Scoping of Variables

- Variables keep values; in python a variable is defined when we assign some value to it
  - Some languages require variables to be declared before they can be assigned anything
- Scope of a variable - where the variable can be accessed
- Global variables - those defined outside any function - they are potentially accessible from anywhere in the program, including fns
- Local variables - those defined in a function, including the parameters - only accessible within the function
- Avoid the using global variables in functions - use only local variables - function is independent and can be understood independently without the global context

Quiz – Single Correct

What will be the output of the code ?

A. 1
B. 2
C. 3
D. 5

```
x = 1
def f(x):
    y = x
    return(y+1)
def g(y):
    y = x
    return y+1
y = 2
print(f(y)+g(y))
```

Quiz – Single Correct

What will be the output of the code ?

A. 1
B. 2
C. 3
D. 5

```
x = 1
def f(x):
    y = x
    return(y+1)
def g(y):
    y = x
    return y+1
y = 2
print(f(y)+g(y))
```

Modifying Global Variables

- Global variables can be accessed within a function through scoping rules
- For changing a global var in a function, scoping not sufficient
- Python requires any global variable to be changed in a function to be explicitly declared as global in the function

```
global X # there must be a variable in main program named X

# X now will refer to the global variable X
```

- Accessing/modifying global variables within a function is to be strictly avoided (only to be used rarely)

Local and Global Variables

```
print("Red")
def f(a, b):
    s = a*b
    print("Green")
    global num1
    num1 = 100
    print("num1:", num1)
    return s

print("Yellow")
num1 = 10
num2 = 5
ans = f(10, 5)
print("Global num1:", num1)
print("Blue")
```

The value of num1 is now 100

Visualize this code using Pythontutor

Quiz – Single Correct

What will be the output of the code ?

A. 0, 10
B. 10, 20
C. 20, 10
D. Error

```
def f(x):
    global y
    y = 10
    return (y+10)
y = 0
print(f(y), y)
```

Quiz – Single Correct

What will be the output of the code ?

A. 0, 10
B. 10, 20
C. 20, 10
D. Error

```
def f(x):
    global y
    y = y+10
    return (y+10)
y = 0
print(f(y), y)
```

Main program with functions

- With functions, most of the computation should be done in the functions; the main program should have minimal computations
- A common way to structure the overall code
  - Define functions for units of computation with clean interfaces - i.e. a few parameters, local vars, and some return value
  - The main program is used mostly for: getting inputs, calling functions to do the processing, and then printing/processing the results
  - In complex programs, you may even have functions for input/output
- Now whenever you write a program, use functions liberally to do most of the computation
- Lets see an example - program to compute factorial

Example – Factorial

```
# A function to take in a number and print its factorial
def factorial(n):
    fact = 1
    for i in range(1,n+1):
        fact = fact*i
    return fact

# Main Program
n = int(input("Enter an integer: "))
if n<=0:
    print("Number is <= 0")
else:
    fact = factorial(n)
    print("Factorial is: ", fact)
```

Visualizing Execution using Pythontutor

- Helps in visualizing what is happening in the program
- Aids understanding of the working / running
- Can use for debugging
- For small programs - learning a construct or a new feature
- Let us run this program

```
def fn(x,y):
    c = x+y
    return c
a = 5
b = 7
d = fn(a, b)
print(d)
```

And the factorial program

Exercise - compute n Choose r

```
# Compute n choose r using the formula
# can use the old function code

# Main Program
n = int(input("Enter n: "))
r = int(input("Enter r: "))
if n<=r:
    print("0 combinations")
else:
    ncr = factorial(n)/(factorial(n-r)*factorial(r))
    print("n choose r is: ", ncr)
```

Examples

```
# Fn to find period of pendulum
def pendulum_period(len):
    g = 9.8
    pi = 3.14
    period = 2*pi*((len/g) ** 0.5)
    return period

# Main Program
l = 2.4
ans = pendulum_period(l)
print(ans)
```

```
# Fn to compute simple interest
def simple_interest(principal,
rate, time):
    interest = principal * rate
* time
    return interest

# Main Program
P = 1000
R = 0.05 # Rate of interest 5%
T = 5

SI = simple_interest(P, R, T)
print(SI)
```

Example – HCF

```
def hcf(a,b):
    if a > b:
        smaller = b
    else:
        smaller = a
    for i in range(1, smaller + 1):
        if a % i == 0 and b % i == 0:
            hcf = i
    return hcf

# Main Program
a = 54
b = 12
print("The H.C.F. of", a, "and", b, "is", hcf(a,b))
```

Example – Roots of Quadratic Polynomial

```
# Calculates the solutions to the quadratic a*x**2 + b*x + c = 0
def quadratic_roots(a, b, c):
    # Calculate the discriminant
    d = (b**2) - (4*a*c)
    # Test if discriminant is negative
    if d < 0:
        return None
    else:
        # Calculate the two roots
        x1 = (-b + d**0.5)/(2*a)
        x2 = (-b - d**0.5)/(2*a)
        return (x1, x2)

# Main Program
a, b, c = 1, -5, 6
print("The solutions are:", quadratic_roots(a, b, c))
```

Quiz

You have to write a function to compute LCM of two numbers. Give its header definition - use the name lcm for the function, and if you need any parameters use a, b, c, ….

Importance of Functions

- Functions are a powerful tool for writing large programs
  - **Divide and conquer** - allows the large programming problem to be divided into smaller problems, with functions written for solving sub-problems, then combined to solve the problem
  - **Abstraction** - encapsulate a computation to be used anywhere by just using the function name; don't have to understand function logic for using (it may be written by someone else)
  - **Reusability** - the same function can be called from many places, i.e. the function code is being reused many times
  - **Modularity** - with function, a program is a set of functions (modules) multiple of these are connected together for building a solution
- Functions are the oldest method in programming languages for providing modularity, abstraction, etc
  - Even the earliest languages provided this abstraction

Defining Functions – Some Practices

- A function must have an expressive name which represents what the function is doing
  - Generally names start with lower case letter
- Should have a clean and simple interface - with few parameters
- Should be computing something that can be easily stated in a simple sentence
- Should not have any side effects - i.e. caller only gets returned values, no other changes in any vars in caller or main program
- Must have a comment - which states succinctly ***what*** the program is doing (not its logic) - is also a test of whether the function has a clean abstraction / purpose
- Naming standards - PEP 8 has conventions followed widely

More about Using Fns for Modularity

- Use of functions in the program promotes modularity and code reusability.
- Modular programming emphasise on subdividing a computer program into separate sub-programs (functions) to increase the maintainability, readability of the code and to make it easier to introduce any changes in future or to correct the errors
  - Always have a comment describing what the function is doing (not how) - helps making it modular
  - If you have to write a long commentary to explain - rethink
- A function can be defined once and used multiple times in the program. This reduces the lines of code that the programmer needs to write.
- The significance of functions becomes clear when the size of program becomes large.