Python –S4

Contents

Functions

Functions

- In the context of programming, a function is a named sequence of statements that performs a computation.
- When you define a function, you specify the name and the sequence of statements.
- Later, you can “call” the function by name.
- Consider a simple (built-in) function :

```
>>> type (32)
<class ‘int’>
```

- The name of the function is type.
- The expression in parentheses is called the argument of the function.
- The argument is a value or variable that we are passing into the function as input to the function.
- The result, for the type function, is the type of the argument. [class ‘int’]
- It is common to say that a function “takes” an argument and “returns” a result. The result is called the return value.

Functions

Built-in Functions

- Python provides a number of important built-in functions that we can use without needing to provide the function definition.
- The max and min functions give us the largest and smallest values in a list, respectively:

```
>>> max('Hello world')
'w'
>>> min('Hello world')
' '
>>>
```

- The max function tells us the "largest character" in the string (which turns out to be the letter "w") and the min function shows us the smallest character which turns out to be a space.

Functions

Built-in Functions

- Another very common built-in function is the len function which tells us how many items are in its argument.
- If the argument to len is a string, it returns the number of characters in the string. E.g.

```
>>> len('Hello world')
11
>>>
```

- These functions are not limited to looking at strings, they can operate on any set of values.
- You should treat the names of built-in functions as reserved words (i.e. avoid using “max” as a variable name).

Functions

Type Conversion Functions

- Python also provides built-in functions that convert values from one type to another.
- For example : int can convert floating-point values to integers, but it doesn't round off; it chops off the fraction part:

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

- float converts integers and strings to floating-point numbers:

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

- The function str converts its arguments to a string :

```
>>> str(32)
‘32’
```

Functions

Random Numbers

- The random module provides functions that generate pseudorandom numbers.
- The function random returns a random float between 0.0 and 1.0 (including 0.0 but not 1.0).
- Each time you call random, you get the next number in a long series. To see a sample, run this loop:

```
import random
for i in range(5):
    x = random.random()
    print (x)
```

- It will print 5 random numbers between 0 and 1.

- Note : You need to import certain libraries for Built-in functions

Functions

Random Numbers

- Let's take another example:

```
import random
for i in range(5):
    x = random.randint(0,10)
    print (x)
```

- It will print 5 random numbers between 0 and 10.
- So

random.randint(a, b)

- Return a random integer *N* such that a <= N <= b

Functions

Math Functions

- Python has a math module that provides most of the familiar mathematical functions.
- Before we can use the module, we have to import it:

```
>>> import math
```

- This statement creates a module object named math.
- To access one of the functions, you have to specify the name of the module and the name of the function, separated by a dot (also known as a period).
- This format is called dot notation.

```
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)

>>> radians = 0.7
>>> height = math.sin(radians)
```

Functions

Math Functions

- Let us see some more math functions in Python 3
- math.factorial(*x*) -- Return *x* factorial. Raises ValueError if *x* is not integral or is negative.
- math.exp(*x*) -- Return e**x.
- math.log(*x*[, *base*]) -- Return the logarithm of *x* to the given *base*. If the *base* is not specified, return the natural logarithm of *x* (that is, the logarithm to base *e*).
- math.log10(*x*) -- Return the base-10 logarithm of *x*.
- math.pow(*x*, *y*) -- Return x raised to the power y. Note : 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.
- math.sqrt(*x*) -- Return the square root of *x*.

Functions

Math Functions

- math.acos(*x*) Return the arc cosine of *x*, in radians.
- math.asin(*x*) Return the arc sine of *x*, in radians.
- math.atan(*x*) Return the arc tangent of *x*, in radians.
- math.cos(*x*) Return the cosine of *x* radians.
- math.hypot(*x*, *y*) Return the Euclidean norm, sqrt(x*x + y*y). This is the length of the vector from the origin to point (x, y).
- math.sin(*x*) Return the sine of *x* radians.
- math.tan(*x*) Return the tangent of *x* radians.

- math.degrees(*x*) Converts angle *x* from radians to degrees.
- math.radians(*x*) Converts angle *x* from degrees to radians.

For complete list --- see Python 3.x Documentation

Functions

Some other Built-in Functions

- abs() – Return the absolute value of a number
- chr() – Return the string representing a character
- help() – Invoke the built-in help system
- pow() – Return power raised to a number
- float() − Convert a string or a number to floating point
- hash(object) − return the hash value of the object
- range(start, stop[, step]) − the arguments must be integers.
  - If the *step* argument is omitted, it defaults to 1. If the *start* argument is omitted, it defaults to 0.
  - For a positive *step*, the contents of a range r are determined by the formula r[i] = start + step*i where i >= 0 and r[i] < stop.
  - For a negative *step*, the contents of the range are still determined by the formula r[i] = start + step*i, but the constraints are i >= 0 and r[i] > stop.
- round(*number*[, *ndigits*]) -- return the floating point value *number* rounded to *ndigits* digits after the decimal point. If *ndigits* is omitted, it defaults to zero.

Functions

Some other Built-in Functions

- dir() with an argument -- This function is extremely useful for discovering what methods or attributes an object has. For example,

```
>>> mystr = “hello”
>>> dir(mystr)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__',
'__getslice__', '__gt__', '__hash__', '__init__', '__le__', '__len__', '__lt__',
'__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '_formatter_field_name_split', '_formatter_parser',
'capitalize', 'center', 'count', 'decode', 'encode', 'endswith', 'expandtabs',
'find', 'format', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace', 'istitle',
'isupper', 'join', 'ljust', 'lower', 'lstrip', 'partition', 'replace', 'rfind', 'rindex',
'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip',
'swapcase', 'title', 'translate', 'upper', 'zfill']
```

Functions

Some other Built-in Functions

- Please note that dir() function will give you a different set of methods available depending on the value of the variable. Say for int:

```
>>> mynum = 5
>>> dir(mynum)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__', '__floor__', '__floordiv__', '__format__', '__ge__', '__getattribute__', '__getnewargs__', '__gt__', '__hash__', '__index__', '__init__', '__init_subclass__', '__int__', '__invert__', '__le__', '__lshift__', '__lt__', '__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__or__', '__pos__', '__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__', '__ror__', '__round__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__', '__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__', '__subclasshook__', '__truediv__', '__trunc__', '__xor__', 'bit_length', 'conjugate', 'denominator', 'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']
```

Functions

Writing your own Functions

Functions

Writing your own Functions

- A function definition specifies the name of a new function and the sequence of statements that execute when the function is called.
- Once we define a function, we can reuse the function over and over throughout our program.
- Here is an example:

```
def print_hello():
    print (”Hello World." )
    print (’My Name is Prasun’)
```

- def is a keyword that indicates that this is a function definition.
- The name of the function is print_hello.

Functions

Writing your own Functions

- Let us consider some more simple Functions :

```
def hello():
    print("Hello")

def area(width, height):
    return width * height

def print_welcome(name):
    print("Welcome", name)
```

Functions

Writing your own Functions

- The rules for function names are the same as for variable names: letters, numbers and some punctuation marks are legal, but the first character can't be a number.
- You can't use a keyword as the name of a function, and you should avoid having a variable and a function with the same name.
- The empty parentheses after the name indicate that this function doesn't take any arguments.
- Later we will build functions that take arguments as their inputs.
- The first line of the function definition is called the header; the rest is called the body.
- The header has to end with a colon and the body has to be indented.
- By convention, the indentation is always four spaces.
- The body can contain any number of statements.

Functions

Writing your own Functions

- If you type a function definition in interactive mode, the interpreter prints ellipses (...) to let you know that the definition isn't complete:

```
>>> def print_lyrics():
...     print ("I'm a lumberjack, and I'm okay." )
...     print ('I sleep all night and I work all day.' )
...
```

- To end the function, you have to enter an empty line (this is not necessary in a script).
- Defining a function creates a variable with the same name.
- The syntax for calling the new function is the same as for built-in functions:

```
>>> print_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

- If a function exists, you can use it within another function.

Functions

Parameters and Arguments

- Some of the built-in functions we have seen require arguments.
- For example, when you call math.sin you pass a number as an argument.
- Some functions take more than one argument: math.pow takes two, the base and the exponent.
- Inside the function, the arguments are assigned to variables called parameters.
- Here is an example of a user-defined function that takes an argument:

```
def print_twice(arg1):
    print (arg1 )
    print (arg1)
```

- This function assigns the argument to a parameter named bruce.
- When the function is called, it prints the value of the parameter (whatever it is) twice.

Functions

Parameters and Arguments

- This function works with any value that can be printed.

```
>>> print_twice('Spam')
Spam
Spam

>>> print_twice(math.pi)
3.14159265359
3.14159265359
```

- You can also use a variable as an argument:

```
>>> text = ‘My name is Krishna’
>>> print_twice(text)
My name is Krishna
My name is Krishna
```

Local and global variables

- To understand local and global variables we will go through two examples.

```
def change(b):
    a = 90
    print(a)
a=9
print("Before the function call ", a)
print("inside change function", end=' ')
change(a)
print("After the function call ", a)
```

- The output will be :

```
Before the function call 9
inside change function 90
After the function call 9
```

Local and global variables

- The next example :

```
def change(b):
    global a
    a = 90
    print(a)
a=9
print("Before the function call ", a)
print("inside change function", end=' ')
change(a)
print("After the function call ", a)
```

- The output will be :

```
Before the function call 9
inside change function 90
After the function call 90
```

Default argument value

- In a function variables may have default argument values

```
def test (a, b =  -99):
        if a > b:
                return True
        else:
                return False
```

- In the above example we have written *b = -99* in the function parameter list. That means if no value for *b* is given then b's value is *-99*

```
>>> test(12,23)
False
>>>test(12)
True
```

Note : Remember that you can not have an argument without default argument if you already have one argument with default values before it. Like *f(a, b=90, c)* is illegal as *b* is having a default value but after that *c* is not having any default value.

Keyword arguments

- Now consider:

```
def  func(a, b=5, c=10):
       print(‘a is ‘, a, ‘ and b is ‘, b, ‘ and c is ‘, c)
```

- Now if we call the function as :

```
>>> func(12, 24)
a is 12 and b is 24 and c is 10

>>> func(12, c = 24)
a is 12 and b is 5 and c is 24

>>> func(b=12, c = 24, a = -1)
a is -1 and b is 12 and c is 24
```

The return statement

- The statement return [expression] exits a function, optionally passing back an expression to the caller.
- A return statement with no arguments is the same as return None.
- return statement in Python function is optional

```
# Function definition is here
def sum( arg1, arg2 ):
    # Add both the parameters and return them."
    total = arg1 + arg2
    return total

# Now you can call sum function
total = sum( 10, 20 )
print ( total )
```

Multiple Function Arguments

- Every function in Python receives a predefined number of arguments, if declared normally, like this:

```
def myfunction(first, second, third):
    # do something here
```

- It is possible to declare functions which receive a variable number of arguments, using the following syntax:

```
def myfunction2(first, second, third, *therest):
    print("First: %s" % first)
    print("Second: %s" % second)
    print("Third: %s" % third)
    print("And all the rest... %s" % list(therest))
```

Multiple Function Arguments

- Now if we call the function myfunc2 as

myfunction2(1,2,3,4,5)

- The output becomes :

```
First: 1
Second: 2
Third: 3
And all the rest... [4, 5]
```

- The "therest" variable is a list of variables, which receives all arguments which were given to the ”myfinction2" function after the first 3 arguments.

Why Functions

- It may not be clear why it is worth the trouble to divide a program into functions.
- There are several reasons:

- ❖ Creating a new function gives you an opportunity to name a group of statements, which makes your program easier to read, understand and debug.
- ❖ Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place.
- ❖ Dividing a long program into functions allows you to debug the parts one at a time and then assemble them into a working whole.
- ❖ Well-designed functions are often useful for many programs.
- ❖ Once you write and debug one, you can reuse it.

Functions – Example programs

```
# C to F or F to C converter
def  ftoc():
        f = int(input("Enter a temperature in Fahrenheit : "))
        c = (f - 32) * 5.0/9.0
        print("Temperature :", f, " Fahrenheit is == ", c, " Celsius")

def ctof():
        c = int(input("Enter a temperature in Celcius : "))
        f = 9.0/5.0 * c + 32
        print("Temperature :", c, " Celcius is == ", f, " Fahrenheit")

choice = input("Enter your choice : ctof or ftoc  ==> ")
while choice not in ['ctof', 'ftoc' ]:
        print("Incorrect option……..!!")
        choice = input("Enter your choice : ctof or ftoc  ==> ")

if choice == 'ctof':
        ctof()
else:
        ftoc()
```

Functions – Example programs

```
# calculates a given rectangle area
def area(width, height):
    return width * height
def positive_input(prompt):
    number = float(input(prompt))
    while number <= 0:
        print('Must be a positive number')
        number = float(input(prompt))
    return number

print('To find the area of a rectangle,')
print('enter the width and height below.')
print()
w = positive_input('Width: ')
h = positive_input('Height: ')
print('Width =', w, ' Height =', h, ' so Area =', area(w, h))
```

Functions- Examples

Let us consider another example :

```
# Make a simple calculator that can add, subtract, multiply and divide using functions
# define functions
def add(x, y):
	‘’’This function adds two numbers‘’’
	return x + y
def subtract(x, y):
	‘’’This function subtracts two numbers‘’’
	return x - y
def multiply(x, y):
	‘’’This function multiplies two numbers‘’’
	return x * y
def divide(x, y):
	‘’’This function divides two numbers‘’’
	return x / y
```

[contd.]

Functions – Examples

```
# take input from the user
print("Select operation.")
print("1.Add")
print("2.Subtract")
print("3.Multiply")
print("4.Divide")
choice = input("Enter choice(1/2/3/4):")
num1 = int(input("Enter first number: "))
num2 = int(input("Enter second number: "))
if choice == '1':
        print(num1,"+",num2,"=", add(num1,num2))
elif choice == '2':
        print(num1,"-",num2,"=", subtract(num1,num2))
elif choice == '3':
        print(num1,"*",num2,"=", multiply(num1,num2))
elif choice == '4':
        print(num1,"/",num2,"=", divide(num1,num2))
 else:
        print("Invalid input")
```

Functions – Examples

Example of a Polymorphic function

```
def doubleIt(x):
        return (2 * x)

y = 3
print (doubleIt(y))
```

- The output is : 6

```
z = "Spam "
print (doubleIt(z))
```

- Here the output is : SpamSpam

```
# This program works because the * operator can be used with
# numbers and with strings. This is an example of Polymorphism.

# Poly means "many" and morph means "form"

# Polymorphism : the meaning of the operations depends on the objects
# being operated on. The * operator is said to be "overloaded"
# An overloaded operator behaves differently depending on
# the type of its operands.
```

Functions – Examples

```
## an interesting example
def pause():
    input("\n\nPress any key to continue...\n\n")
def quitMessage():
    print ("Thank you for using this program\n Goodbye")
def printThreeLines():
    for i in range(1,4):
        print ('this is line ' + str(i))
def printNineLines():
    for i in range(1,4):
        printThreeLines()
def startMessage():
    print ("This program demonstrates the use of Python functions")
    pause()
def blank_Line():
    print()
def clearScreen():
    for i in range(1,26):
        blank_Line()
```

Functions – Examples

```
## an interesting example   [ contd. ]

startMessage()
clearScreen()
print ("Testing this program")
printNineLines()
pause()
clearScreen()
 printNineLines()
blank_Line()
printNineLines()
pause()
clearScreen()
quitMessage()
```

Functions – Examples

```
## comparing two dna sequences
seq_a = "TGGAGGCAATGGCGGCCAGCA"
seq_b = "GACTCCTCCTCCTCCTGCTCA"
len_a = len(seq_a)
len_b = len(seq_b)
print('Sequence A is : ', seq_a)
print("Length of Sequence A: " + str(len_a))
print()
print('Sequence B is : ', seq_b)
print("Length of Sequence B: " + str(len_b))
```

[contd.]

Functions – Examples

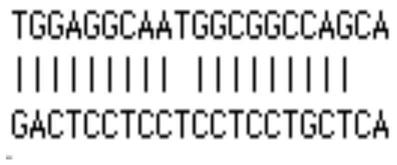
```
## comparing two dna sequences [ contd. ]

def sequence_compare(seq_a, seq_b):
        len1 = len(seq_a)
        len2 = len(seq_b)
        mismatches = []
        for pos in range (0, min(len1, len2)) :
              if seq_a[pos] != seq_b[pos]:
                  mismatches.append('|')
              else:
                  mismatches.append(' ')
        print (seq_a)
        print (''.join(mismatches))
        print (seq_b)

sequence_compare(seq_a, seq_b)
```

Functions – Examples

The output of the previous program would be :



```
TGGAGGCAATGGCGGCCAGCA
||||||||| |||||||||
GACTCCTCCTCCTCCTGCTCA
```

Common Coding Issues[mistakes]

[IMPORTANT]

- Don’t forget the colons
- Start in column 1.
- Blank lines matter at the interactive prompt.
- Indent consistently.
- Don’t code C in Python.
- Use simple for loops instead of while or range.
- Beware of mutables in assignments.
- Don’t expect results from functions that change objects in place.
- Always use parentheses to call a function.
- Don’t use extensions or paths in imports and reloads.

Reference

- Python for Informatics – C. Severance
- Think Python – A. B. Downey - [O'Reilly]
- Python Crash Course – Eric Matthes [No starch Press]
- A Byte of Python – Swaroop C H
- Introducing Python – Bill Lubanovic [O'Reilly]

- Learning Python – 5th Ed – Mark Lutz - [O'Reilly]

End of Presentation

Python – S4