3 Python Basics

"Well begun is half done..."

Contents

- Identifiers and Keywords
- Python Types
- Basic Types
- Integer and Float Ranges
- Variable Type and Assignment
- Arithmetic Operators
- Operation Nuances
- Precedence and Associativity
- Conversions
- Built-in Functions
- Built-in Modules
- Container Types
- Python Type Jargon
- Comments and Indentation
- Multi-lining
- Classes and Objects
- Multiple Objects
- Programs
- Exercises

KanNotes

Identifiers and Keywords

- Python is a case sensitive language.
- Python identifier is a name used to identify a variable, function, class, module, or other object.
- Rules for creating identifiers:
  - Starts with alphabet or an underscore.
  - Followed by zero or more letters, _ , and digits.
  - keyword cannot be used as identifier.
- All keywords are in lowercase.
- Python has 33 keywords shown in Figure 3.1.

False	continue	from	not
None	def	global	or
True	del	if	pass
and	elif	import	raise
as	else	in	return
assert	except	is	try
break	finally	lambda	while
class	for	nonlocal	with
yield			

Figure 3.1

- You can print a list of Python keywords through the statements:

```
import keyword              # makes the module 'keyword' available
print(keyword.kwlist)       # syntax modulename.object/function
```

Python Types

- Python supports 3 categories of data types:

  Basic types - int, float, complex, bool, string, bytes
  Container types - list, tuple, set, dict
  User-defined types - class

- Out of these, basic types will be covered in this chapter in detail. Container types will be covered briefly. A separate chapter is dedicated to each container type, where they are covered in great detail. User-defined types will not be covered in this chapter. Chapter 17 discusses how to create and use them.

Basic Types

- Examples of different basic types are given below:

```
# int can be expressed in binary, decimal, octal, hexadecimal
# binary starts with 0b/0B, octal with 0o/0O, hex with 0x/0X
0b10111, 156, 0o432, 0x4A3

# float can be expressed in fractional or exponential form
- 314.1528, 3.141528e2, 3.141528E2

# complex contains real and imaginary part
3 + 2j, 1 + 4J

# bool can take any of the two Boolean values both starting in caps
True, False

# string is an immutable collection of Unicode characters enclosed
# within ' ', " " or """ """.
'Razzmatazz', "Razzmatazz", """Razzmatazz"""

# bytes represent binary data
b'\xa1\xe4\x56'   # represents 3 bytes with hex values a1a456
```

- Type of particular data can be checked using a function called **type( )** as shown below:

```
print(type(35))          # prints <class 'int'>
print(type(3.14))        # prints <class 'float'>
```

Integer and Float Ranges

- **int** can be of any arbitrary size

```
a = 123
b = 1234567890
c = 12345678901234567890123456789O
```

Python has arbitrary precision integers. Hence you can create as big integers as you want. Moreover, arithmetic operations can be performed on integers without worrying about overflow/underflow.

- Floats are represented internally in binary as 64-bit double-precision values, as per the IEEE 754 standard. As per this standard, the maximum value a float can have is approximately 1.8 x 10^{308}. A number greater than this is represented as **inf** (short for infinity).
- Many floats cannot be represented 'exactly' in binary form. So the internal representation is often an approximation of the actual value.
- The difference between the actual value and the represented value is very small and should not usually cause significant problems.

Variable Type and Assignment

- There is no need to define type of a variable. During execution the type of the variable is inferred from the context in which it is being used. Hence Python is called dynamically-typed language.

```
a = 25          # type of a is inferred as int
a = 31.4        # type of a is inferred as float
a = 'Hi'        # type of a is inferred as str
```

- Type of a variable can be checked using the built-in function **type( )**.

```
a = 'Jamboree'
print(type(a))   # type will be reported as str
```

- Simple variable assignment:

```
a = 10
pi = 3.14
name = 'Sanjay'
```

- Multiple variable assignment:

```
a = 10 ; pi = 31.4 ; name = 'Sanjay'   # use ; as statement separator
a, pi, name = 10, 3.14, 'Sanjay'
a = b = c = d = 5
```

Arithmetic Operators

- Arithmetic operators: + - * / % // **

```
a = 4 / 2       # performs true division and yields a float 2.0
a = 7 % 2       # % yields remainder 1
```

```
b = 3 ** 4        # ** yields 3 raised to 4 (exponentiation)
c = 4 // 3        # // yields quotient 1 after discarding fractional part
```

- In-place assignment operators offer a good shortcut for arithmetic operations. These include += -= *= /= %= //= **=.

```
a **= 3           # same as a = a ** 3
b %= 10           # same as b = b % 10
```

Operation Nuances

- On performing floor division **a // b**, result is the largest integer which is less than or equal to the quotient. **//** is called floor division operator.

```
print(10 // 3)      # yields 3
print(-10 // 3)     # yields -4
print(10 // -3)     # yields -4
print(-10 // -3)    # yields 3
print(3 // 10)      # yields 0
print(3 // -10)     # yields -1
print(-3 // 10)     # yields -1
print(-3 // -10)    # yields 0
```

In -10 // 3, multiple of 3 which will yield -10 is -3.333, whose floor value is -4.

In 10 // -3, multiple of -3 which will yield 10 is -3.333, whose floor value is -4.

In -10 // -3, multiple of -3 which will yield -10 is 3.333, whose floor value is 3.

- **print( )** is a function which is used for sending output to screen. Iy can be used in many forms. They are discussed in Chapter 7.

- Operation **a % b** is evaluated as **a - (b * (a // b))**. This can be best understood using the following examples:

```
print(10 % 3)       # yields 1
print(-10 % 3)      # yields 2
print(10 % -3)      # yields -2
print(-10 % -3)     # yields -1
print(3 % 10)       # yields 3
print(3 % -10)      # yields -7
```

```
print(-3 % 10)      # yields 7
print(-3 % -10)     # yields -3
```

Since a % b is evaluated as a - (b * (a // b)),
-10 % 3 is evaluated as -10 - (3 * (-10 // 3)), which yields 2
10 % -3 is evaluated as 10 - (-3 * (10 // -3)), which yields -2
-10 % -3 is evaluated as -10 - (-3 * (-10 // -3)), which yields -1

- Mathematical rule **a / b x c** is same as **a x c / b** holds, but not always.

```
# following expressions give same results
a = 300 / 100 * 250
a = 300 * 250 / 100
```

```
# However, these don't
b = 1e210 / 1e200 * 1e250
b = 1e210 * 1e250 / 1e200      # gives INF
```

- Since True is 1 and False is 0, they can be added.

```
a = True + True         # stores 2
b = True + False        # stores 1
```

Precedence and Associativity

- When multiple operators are used in an arithmetic expression, it is evaluated on the basis of precedence (priority) of the operators used.
- Operators in decreasing order of their priority (PEMDAS):

```
( )             # Parentheses
**              # Exponentiation
*, /, //, %     # Multiplication, Division
+, -            # Addition, Subtraction
```

- If there is a tie between operators of same precedence, it is settled using associativity of operators.
- Each operator has either left to right associativity or right to left associativity.
- In expression c = a * b / c, * is done before / since arithmetic operators have left to right associativity.

- A complete list of Python operators, their priority and associativity is given in Appendix A.

Conversions

- Mixed mode operations:
  - Operation between **int** and **float** will yield **float**.
  - Operation between **int** and **complex** will yield **complex**.
  - Operation between **float** and **complex** will yield **complex**.
- We can convert one numeric type to another using built-in functions **int( )**, **float( )**, **complex( )** and **bool( )**.
- Type conversions:

```
int(float/numeric string)    # from float/numeric string to int
int(numeric string, base)    # from numeric string to int in base

float(int/numeric string)    # from int/numeric string to float
float(int)                   # from int to float

complex(int/float)   # convert to complex with imaginary part 0
complex(int/float, int/float)   # convert to complex

bool(int/float)          # from int/float to True/False (1/0)
str(int/float/bool)      # converts to string
chr(int)                 # yields character corresponding to int
```

- **int( )** removes the decimal portion from the quotient, so always rounds towards zero.

```
int(3.33)     # yields 3
int(-3.33)    # yields -3
```

Built-in Functions

- Python has many built-in functions that are always available in any part of the program. The **print( )** function that we have been using to send output to screen is a built-in function.
- Help about any built-in function is available using **help(function)**.
- Built-in functions that are commonly used with numbers are given below:

```
abs(x)          # returns absolute value of x
pow(x, y)       # returns value of x raised to y
min(x1, x2,...) # returns smallest argument
```

```
max(x1, x2,...)     # returns largest argument
divmod(x, y)        # returns a pair(x // y, x % y)
round(x [,n])       # returns x rounded to n digits after .
bin(x)              # returns binary equivalent of x
oct(x)              # returns octal equivalent of x
hex(x)              # returns hexadecimal equivalent of x
```

- Following Python program shows how to use some of these built-in functions:

```
a = abs(-3)                 # assigns 3 to a
print(min(10, 20, 30, 40))  # prints 10
print(hex(26))              # prints 1a
```

Built-in Modules

- Apart from built-in functions, Python provides many built-in modules. Each module contains many functions.

- For performing sophisticated mathematical operations we can use the functions present in built-in modules **math**, **cmath**, **random**, **decimal.**

  math - many useful mathematics functions.
  cmath - functions for performing operations on complex numbers.
  random - functions related to random number generation.
  decimal - functions for performing precise arithmetic operations.

- Mathematical functions in **math** module:

```
pi, e           # values of constants pi and e
sqrt(x)         # square root of x
factorial(x)    # factorial of x
fabs(x)         # absolute value of float x
log(x)          # natural log of x (log to the base e)
log10(x)        # base-10 logarithm of x
exp(x)          # e raised to x
trunc(x)        # truncate to integer
ceil(x)         # smallest integer >= x
floor(x)        # largest integer <= x
modf(x)         # fractional and integer parts of x
```

- **round( )** built-in function can round to a specific number of decimal places, whereas **math** module's library functions **trunc( )**, **ceil( )** and **floor( )** always round to zero decimal places.

- Trigonometric functions in **math** module:

```
degrees(x)      # radians to degrees
radians(x)      # degrees to radians
sin(x)          # sine of x radians
cos(x)          # cosine of x radians
tan(x)          # tan of x radians
sinh(x)         # hyperbolic sine of x
cosh(x)         # hyperbolic cosine of x
tanh(x)         # hyperbolic tan of x
acos(x)         # cos inverse of x, in radians
asin(x)         # sine inverse of x, in radians
atan(x)         # tan inverse of x, in radians
hypot(x, y)     # sqrt(x * x + y * y)
```

- Random number generation functions from **random** module:

```
random( )               # random number between 0 and 1
randint(start, stop)    # random number in the range
seed( )  # sets current time as seed for random number generation
seed(x)  # sets x as seed for random number generation logic
```

- To use functions present in a module, we need to import the module using the **import** statement.

- Following Python program shows how to use some of the functions of **math** module and **random** module:

```
import math
import random
print(math.factorial(5))            # prints 120
print(math.degrees(math.pi))        # prints 180.0
print(random.random( ))             # prints 0.8960522546341796
```

- There are many built-in functions and many functions in each built-in module. It is easy to forget the names of the functions. We can get a quick list of them using the following program:

```
import math
print(dir(__builtins__))      # 2 underscores before and after builtins
```

```
print(dir(math))
```

Container Types

- Container types typically refer to multiple values stored together. Examples of different basic types are given below:

  # list is a indexed collection of similar/dissimilar entities
  [10, 20, 30, 20, 30, 40, 50, 10], ['She', 'sold', 10, 'shells'']

  # tuple is an immutable collection
  ('Sanjay', 34, 4500.55), ('Let Us Python', 350, 195.00)

  # set is a collection of unique values
  {10, 20, 30, 40}, {'Sanjay', 34, 45000}

  # dict is a collection of key-value pairs, with unique key enclosed in ' '
  {'ME101' : 'Strength of materials', 'EE101' : 'Electronics'}

- Values in a list and tuple can be accessed using their position in the list or tuple. Values in a set can be accessed using a **for** loop (discussed in Chapter 6). Values in a dictionary can be accessed using a key. This is shown in the following program:

```
lst = [10, 20, 30, 20, 30, 40, 50, 10]
tpl = ('Let Us Python', 350, 195.00)
s = {10, 20, 30, 40}
dct = {'ME101' : 'SOM', 'EE101' : 'Electronics'}
print(lst[0], tpl[2])       # prints 10  195.0
print(dct['ME101'])         # prints SOM
```

Python Type Jargon

- Often following terms are used while describing Python types:

  Collection - a generic term for container types.

  Iterable - means a collection that can be iterated over using a loop.

  Ordered collection - elements are stored in the same order in which they are inserted. Hence its elements can be accessed using an index, i.e. its position in the collection.

  Unordered collection - elements are not stored in the same order in which they are inserted. So we cannot predict at which position a particular element is present. So we cannot access its elements using a position based index.

Sequence is the generic term for an ordered collection.

Immutable - means unchangeable collection.

Mutable - means changeable collection.

- Let us now see which of these terms apply to types that we have seen so far.

  String - ordered collection, immutable, iterable.
  List - ordered collection, mutable, iterable.
  Tuple - ordered collection, immutable, iterable.
  Set - unordered collection, mutable, iterable.
  Dictionary - unordered collection, mutable, iterable.

Comments and Indentation

- Comments begin with #.

```
# calculate gross salary
gs = bs + da + hra + ca
si = p * n * r / 100        # calculate simple interest
```

- Multi-line comments should be written in a pair of ''' or """.

```
''' Additional program: Calculate bonus to be paid
   URL: https://www.ykanetkar.com (https://www.ykanetkar.com)
   Author: Yashavant, Date: 18 May 2020 '''
```

- Indentation matters! Don't use it casually. Following code will report an error 'Unexpected indent'.

```
a = 20
    b = 45
```

Multi-lining

- If statements are long they can be written as multi-lines with each line except the last ending with a \.

```
total = physics + chemistry + maths + \
    english + Marathi + history + \
    geography + civics
```

- Multi-line statements within [ ], { }, or ( ) don't need \.

```
days = [ 'Monday', 'Tuesday', 'Wednesday', Thursday',
         'Friday', 'Saturday', 'Sunday' ]
```

Classes and Objects

- In Python every type is a class. So **int**, **float**, **complex**, **bool**, **str**, **list**, **tuple**, **set**, **dict** are all classes. These are ready-made classes. Python also permits us to create user-defined classes as we would see in Chapter 18.
- An object is created from a class. A class describes two things—the form an object created from it will take and the methods (functions) that can be used to access and manipulate the object.
- From one class multiple objects can be created. When an object is created from a class, it is said that an instance of the class is being created.
- A class has a name, whereas objects are nameless. Since objects do not have names, they are referred using their addresses in memory.
- All the above statements can be verified through the following program. Refer to Figure 3.1 to understand it better.

```
a = 30
b = 'Good'
print(a, b)                    # prints 3 Good
print(type(a), type(b))        # prints <class 'int'> <class 'str'>
print(id(a), id(b))            # prints  1356658640  33720000
print(isinstance(a, int), isinstance(b, str))    # prints True True
```

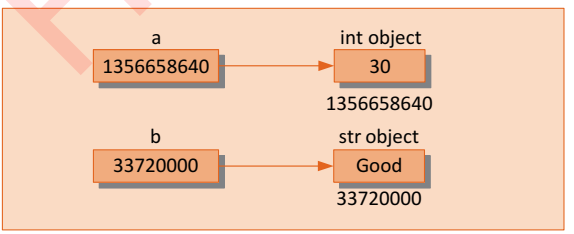

Figure 3.1

- In this program we have created two objects—one from ready-made class **int** and another from ready-made class **str**.

- The object of type **int** contains 30, whereas the object of type **str** contains 'Good'.
- Both the objects are nameless. Their addresses in memory are 1356658640 and 33720000 which are stored in **a** and **b**.
- These addresses can be obtained using the built-in function **id( )**. When you execute the program you may get different addresses.
- Since **a** and **b** contain addresses they are said to refer to objects present at these addresses. In simpler words they are pointers to objects.
- Type of objects to which **a** and **b** are referring to can be obtained using the built-in function **type( )**.
- Whether **a** refers to an instance of class **int** can be checked using the built-in function **instanceof( )**.

Multiple Objects

- Consider the following program:

```
a = 3
b = 3
print(id(a), id(b))      # prints 1356658640  1356658640
print(a is b)            # prints True
a = 30                   # now a refers to a different object
print(id(a))             # prints 1356659072
```

- Are we creating 2 **int** objects? No. Since the value stored in **int** object is same, i.e. 3, only 1 **int** object is created. Both **a** and **b** are referring to the same **int** object. That is why **id(a)** and **id(b)** return same addresses.
- This can also be verified using the **is** operator. It returns True since **a** and **b** both are referring to the same object.
- When we attempt to store a new value in **a**, a new **int** object is created as a different value, 30, is to be stored in it. **a** now starts referring to this new **int** object, whereas **b** continues to refer to **int** object with value 3.
- Instead of saying that **a** is referring to an **int** object containing a value 3, it is often said that **a** is an **int** object, or 3 is assigned to

int object **a**. Many programmers continue to believe that **a** and **b** are **int** variables, which we now know is not the case.

Problem 3.1

Demonstrate use of integer types and operators that can be used on them.

Program

```
# use of integer types
print(3 / 4)
print(3 % 4)
print(3 // 4)
print(3 ** 4)

a = 10 ; b = 25 ; c = 15 ; d = 30 ; e = 2 ; f = 3 ; g = 5
w = a + b - c
x = d ** e
y = f % g
print(w, x, y)
h = 99999999999999999
i = 54321
print(h * i)
```

Output

```
0.75
3
0
81
20 900 3
5432099999999999945679
```

Tips

- 3 / 4 doesn't yield 0.
- Multiple statements in a line should be separated using **;**

- **print(w, x, y)** prints values separated by a space.

Problem 3.2

Demonstrate use of **float**, **complex** and **bool** types and operators that can be used on them.

Program

```
# use of float
i = 3.5
j = 1.2
print(i % j)

# use of complex
a = 1 + 2j
b = 3 *(1 + 2j)
c = a * b
print(a)
print(b)
print(c)
print(a.real)
print(a.imag)
print(a.conjugate( ))
print(a)

# use of bool
x = True
y = 3 > 4
print(x)
print(y)
```


Output

```
1.1
(1+2j)
(3+6j)
(-9+12j)
1.0
2.0
(1-2j)
```

```
(1+2j)
True
False
```

Tips

- % works on floats.
- It is possible to obtain **real** and **imag** part from a complex number.
- On evaluation of a condition it replaced by **True** or **False**.

Problem 3.3

Demonstrate how to convert from one number type to another.

Program

```
# convert to int
print(int(3.14))          # from float to int
a = int('485')            # from numeric string to int
b = int('768')            # from numeric string to int
c = a + b
print(c)
print(int('1011', 2))     # convert from binary to decimal int
print(int('341', 8))      # convert from octal to decimal int
print(int('21', 16))      # convert from hex to decimal int

# convert to float
print(float(35))          # from int to float
i = float('4.85')         # from numeric string to float
j = float('7.68')         # from numeric string to float
k = i + j
print(k)

# convert to complex
print(complex(35))        # from int to float
x = complex(4.85, 1.1)    # from numeric string to float
y = complex(7.68, 2.1)    # from numeric string to float
z = x + y
print(z)
```

```
# convert to bool
print(bool(35))
print(bool(1.2))
print(int(True))
print(int(False))
```

Output

```
3
1253
11
225
33
35.0
12.53
(35+0j)
(12.53+3.2j)
True
True
1
0
```

Tips

- It is possible to convert a binary numeric string, octal numeric string or hexadecimal numeric string to equivalent decimal integer. Same cannot be done for a **float**.
- While converting to complex if only one argument is used, imaginary part is considered to be 0.
- Any non-zero number (int or float) is treated as **True**. 0 is treated as **False**.

Problem 3.4

Write a program that makes use of built-in mathematical functions.

Program

```
# built-in math functions
print(abs(-25))
```

```
print(pow(2, 4))
print(min(10, 20, 30, 40, 50))
print(max(10, 20, 30, 40, 50))
print(divmod(17, 3))
print(bin(64), oct(64), hex(64))
print(round(2.567), round(2.5678, 2))
```

Output

```
25
16
10
50
(5, 2)
0b1000000 0o100 0x40
3 2.57
```

Tips

- **divmod(a, b)** yields a pair **(a // b, a % b)**.
- **bin( )**, **oct( )**, **hex( )** return binary, octal and hexadecimal equivalents.
- **round(x)** assumes that rounding-off has to be done with 0 places beyond decimal point.

Problem 3.5

Write a program that makes use of functions in the math module.

Program

```
# mathematical functions from math module
import math
x = 1.5357
print ( math.pi, math.e)
print(math.sqrt( x))
print(math.factorial(6))
print(math.fabs(x))
print(math.log(x))
print(math.log10(x))
print(math.exp(x))
```

```
print(math.trunc(x))
print(math.floor(x))
print(math.ceil(x))
print(math.trunc(-x))
print(math.floor(-x))
print(math.ceil(-x))
print(math.modf(x))
```

Output

```
3.141592653589793 2.718281828459045
1.2392336341465238
720
1.5357
0.42898630314951025
0.1863063842699079
4.644575595215059
1
1
2
-1
-2
-1
(0.5357000000000001, 1.0)
```

Tips

- **floor( )** rounds down towards negative infinity, **ceil( )** rounds up towards positive infinity, **trunc( )** rounds up or down towards 0.
- **trunc( )** is like **floor( )** for positive numbers.
- **trunc( )** is like **ceil( )** for negative numbers.

Problem 3.6

Write a program that generates float and integer random numbers.

Program

```
# random number operations using random module
import random
```

```
import datetime
random.seed(datetime.time( ))
print(random.random( ))
print(random.random( ))
print(random.randint(10, 100))
```

Output

```
0.23796462709189137
0.5442292252959519
57
```

Tips

- It is necessary to import **random** module.
- If we seed the random number generation logic with current time, we get different random numbers on each execution of the program.
- **random.seed( )** with no parameter also seeds the logic with current time.

Problem 3.7

How will you identify which of the following is a string, list, tuple, set or dictionary?

```
{10, 20, 30.5}
[1, 2, 3.14, 'Nagpur']
{12 : 'Simple', 43 : 'Complicated', 13 : 'Complex'}
"Check it out!"
3 + 2j
```

Program

```
# determine type of data
print(type({10, 20, 30.5}))
print(type([1, 2, 3.14, 'Nagpur']))
print(type({12 : 'Simple', 43 : 'Complicated', 13 : 'Complex'}))
print(type("Check it out!"))
print(type(3 + 2j))
```

Output

```
<class 'set'>
<class 'list'>
<class 'dict'>
<class 'str'>
<class 'complex'>
```

Tips

- **type( )** is a built-in function which can determine type of any data—built-in, container or user-defined.

Exercises

[A] Answer the following questions:

(a) Write a program that swaps the values of variables **a** and **b**. You are not allowed to use a third variable. You are not allowed to perform arithmetic on **a** and **b**.

(b) Write a program that makes use of trigonometric functions available in math module.

(c) Write a program that generates 5 random numbers in the range 10 to 50. Use a seed value of 6. Make a provision to change this seed value every time you execute the program by associating it with time of execution?

(d) Use **trunc( )**, **floor( )** and **ceil( )** for numbers -2.8, -0.5, 0.2, 1.5 and 2.9 to understand the difference between these functions clearly.

(e) Assume a suitable value for temperature of a city in Fahrenheit degrees. Write a program to convert this temperature into Centigrade degrees and print both temperatures.

(f) Given three sides a, b, c of a triangle, write a program to obtain and print the values of three angles rounded to the next integer. Use the formulae:

$a^2 = b^2 + c^2 - 2bc \cos A$, $b^2 = a^2 + c^2 - 2ac \cos B$, $c^2 = a^2 + b^2 - 2ab \cos C$

[B] How will you perform the following operations:

(a) Print imaginary part out of 2 + 3j.
(b) Obtain conjugate of 4 + 2j.
(c) Print decimal equivalent of binary '1100001110'.
(d) Convert a float value 4.33 into a numeric string.
(e) Obtain integer quotient and remainder while dividing 29 with 5.
(f) Obtain hexadecimal equivalent of decimal 34567.
(g) Round-off 45.6782 to second decimal place.
(h) Obtain 4 from 3.556.
(i) Obtain 17 from 16.7844.
(j) Obtain remainder on dividing 3.45 with 1.22.

[C] Which of the following is invalid variable name and why?

BASICSALARY	_basic	basic-hra	#MEAN
group.	422	pop in 2020	over
timemindovermatter	SINGLE	hELLO	queue.
team'svictory	Plot # 3	2015_DDay	

[D] Evaluate the following expressions:

(a) 2 ** 6 // 8 % 2
(b) 9 ** 2 // 5 - 3
(c) 10 + 6 - 2 % 3 + 7 - 2
(d) 5 % 10 + 10 -23 * 4 // 3
(e) 5 + 5 // 5 - 5 * 5 ** 5 % 5
(f) 7 % 7 + 7 // 7 - 7 * 7

[E] Evaluate the following expressions:

(a) min(2, 6, 8, 5)
(b) bin(46)
(c) round(10.544336, 2)
(d) math.hypot(6, 8)
(e) math.modf(3.1415)

[F] Match the following pairs:

a. complex	1. \
b. Escape special character	2. Container type
c. Tuple	3 Basic type
d. Natural logarithm	4. log( )
e. Common logarithmlog10( )	5. log10( )