Entering Expressions into the Interactive Shell

You run the interactive shell by launching IDLE, which you installed with Python in the introduction. On Windows, open the Start menu, select **All Programs ▸ Python 3.8**, and then select **IDLE (Python 3.8)**.

A window with the >>> prompt should appear; that's the interactive shell.

```
>>> 2 + 2
4
```

In Python, 2 + 2 is called an *expression*.

```
>>> 2
2
```

Math Operators from Highest to Lowest Precedence

Operator	Operation	Example	Evaluates to...
**	Exponent	2 ** 3	8
%	Modulus/remainder	22 % 8	6
//	Integer division/floored quotient	22 // 8	2
/	Division	22 / 8	2.75
*	Multiplication	3 * 5	15
-	Subtraction	5 - 2	3
+	Addition	2 + 2	4

The order of operations of Python math operators is similar to that of mathematics (**PEDMAS**). The ** operator is evaluated first; the *, /, // and % operators are evaluated next, from left to right; and the + and – operators are evaluated last (also from left to right). You can use parentheses to override the usual precedence if you need to.

Example:

```
>>> 2 + 3 * 6
20
>>> (2 + 3) * 6
30
>>> 48565878 * 578453
28093077826734
```

```
>>> 2 ** 8
256
>>> 23 / 7
3.2857142857142856
>>> 23 // 7
3
>>> 23 % 7
2
>>> (5 - 1) * ((7 + 1) / (3 - 1))
16.0
>>> 2*3%4
2
>>> 2*(3%4)
6
```

Common Data Types:

Data type	Examples
Integers (int)	-2, -1, 0, 1, 2, 3, 4, 5
Floating-point numbers (float)	-1.25, -1.0, --0.5, 0.0, 0.5, 1.0, 1.25
Strings (str) Text values	'a', 'aa', 'aaa', 'Hello!', '11 cats', “asd”

String Concatenation:

```
>>> 'Alice' + 'Bob'
'AliceBob'
>>> 'Alice' + ‘42’
Alice42
>>> 'Alice' + 42
error
>>> 'Alice' * 5
'AliceAliceAliceAliceAlice'
>>> 'Alice' * 'Bob'
error
>>> 'Alice' * 5.0
Error
```

Data Type Check

```
>>> type (4)
<class ’int’>
>>> type (’4’)
<class ’str’>
>>> type (2.0)
<class 'float'>
```

Data Type Conversion

```
>>> str (4)
’4’
>>> ’5’
’5’
>>> int (’5’)
5
>>> float (2)
2
>>> int(2.3)
2.0
```

Variables and Assignment:

A *variable* is like a box in the computer's memory where you can store a single value.

```
>>> spam = 40
>>> spam
40
>>> eggs = 2
 >>> spam + eggs
42
>>> spam + eggs + spam
82
>>> spam = spam + 2
>>> spam
42
>>> spam = 'Hello'
>>> spam
'Hello'
>>> spam = 'Goodbye'
>>> spam
'Goodbye'
```

Variable Names:

You can name a variable anything as long as it obeys the following rules:

1. Variable names can contain letters, numbers, and the underscore.
2. Variable names cannot contain spaces.
3. Variable names cannot start with a number.
4. Case matters—for instance, temp and Temp are different.
5. Cannot use Python keywords

Valid variable names	Invalid variable names
current_balance	current-balance (hyphens are not allowed)
currentBalance	current balance (spaces are not allowed)
t_sum	t_$um (special characters like $ are not allowed)
_spam	42 (can't begin with a number)
hello	'hello' (special characters like ' are not allowed)
account4	4account (can't begin with a number)
with	from (can't use python keywords)

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

The print() Function

The print function enables a Python program to display textual information to the user.

```
>>> print('Hello world!')
Hello world!

>>> x = 2
>>> print(x)
2
>>> print('The value of x =', x)
The value of x = 2

>>> print('x*x=', x*x)
x*x= 4
```

The input () Function

the input function is used to obtain information from the user.

```
>>> x = input ()
4
>>> x
'4'                                    [ By default, input () function always returns a string]

>>> x = int (input ())
4
```

```
>>> x
4

>>> x=float (input ())
4
>>> x
4.0

>>> x = input ('Please enter an integer value: ')
Please enter an integer value: 4
>>> x
'4'

>>> x = int (input ('Please enter an integer value: '))
Please enter an integer value: 4
>>> x
4
```

N.B. the eval function that attempts to evaluate a string in the same way that the interactive shell would evaluate it.

```
>>> x=eval (input ())
3.0
>>> x
3.0

>>> y=eval (input ())
4
>>> y
4
```

Q: Given the following assignment:
x = 2
Indicate what each of the following Python statements would print.
(a) print("x")
(b) print('x')
(c) print(x)
(d) print ("x + 1")
(e) print ('x' + 1)
(f) print (x + 1)

Boolean Values

the *Boolean* data type has only two values: True and False.

>>> **spam = True**

>>> **spam**

True

Comparison Operators

Comparison operators compare two values and evaluate down to a single Boolean value.

Operator	Meaning
==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

>>> **42 == 42**

True

>>> **42 == 99**

False

>>> **2 != 3**

True

>>> **2 != 2**

False

>>> **42 > 100**

False

>>> **42 < 42**

False

>>> **42 == '42'**

False

Boolean Operators

The three Boolean operators (*and*, *or*, and *not*) are used to compare Boolean values.

'*and*' Operator's Truth Table

Expression	**Evaluates to...**
True and True	True
True and False	False
False and True	False
False and False	False

'*or*' Operator's Truth Table

Expression	**Evaluates to...**
True and True	True
True and False	True
False and True	True
False and False	False

'not' Operator's Truth Table

Expression	**Evaluates to...**
True	False
False	True

>>> **(4 < 5) and (5 < 6)**

True

>>> **(4 < 5) and (9 < 6)**

False

>>> **(1 == 2) or (2 == 2)**

True

FLOW CONTROL

1. *if* statement

An *if* statement's clause will execute if the statement's condition is True. The clause is skipped if the condition is False.

Syntax:

if condition:

    statement (s)/execution

```
>>> x = 9
>>> if x = = 10:
        print('ten')
Output:
(empty)
```

```
>>> x = 10
>>> if x = = 10:
        print('ten')

Output:
ten
```

2. *else* statement

An *if* clause can optionally be followed by an *else* statement. The *else* clause is executed only when the *if* statement's condition is False.

Syntax:

if condition:

    if block

else:

    else block

```
>>> x = 9
>>> if x == 10:
        print('ten')
else:
        print ('not ten')
Output:
not ten
```

```
>>> x = 10
>>> if x == 10:
        print('ten')
else:
        print ('not ten')
Output:
ten
```

3. *elif* Statements

The *elif* statement is an "else if" statement that always follows an if or another *elif* statement. It provides another condition that is checked only if any of the previous conditions were False.

Syntax:

```
if condition1:
        if block
elif condition2:
        elif block
else:
        else block
```

```
if grade>=90:
        print('A')
elif grade>=80:
        print('B')
elif grade>=70:
        print('C'):
elif grade>=60:
        print('D'):
else:
        print('F')
```

4. Nested conditions:

If there are conditions within conditions, then in computer programming it is known as nesting.

```
num = int (input ('Enter a number:'))
if num>=0:
        if num = = 0:
                print('zero')
        else:
                print ('Positive number')
else:
        print ('Negative number')
```

Shortcut operators

count=count+1	=>	count+=1
total=total-5	=>	total-=5
prod=prod*2	=>	prod*=2

Iteration

Iteration repeats the execution of a sequence of code.

1. *while* loop Statement:

Syntax:

while condition**:**

block

```
n = 1
while n <= 5:
        print(n)
        n += 1
```

Output:

1

2

3

4

5

2. for loop statement:

The while loop keeps looping while its condition is True, but what if you want to execute a block of code only a certain number of times, then for loop is used.

Syntax:

for *variable name* in range (begin, end, step)**:**

statements to be repeated

```
for n in range (1, 6):
        print(n)
```

where

• begin is the first value in the range; if omitted, the default value is 0

• end is one past the last value in the range; the end value may not be omitted

• change is the amount to increment or decrement; if the change parameter is omitted, it defaults to 1 (counts up by ones)

```
for n in range (21, 0, -3):
        print (n)
```

Output:

21

18

15

12

9

6

3

The following examples show how range can be used to produce a variety of sequences:

• range (10) => 0,1,2,3,4,5,6,7,8,9
• range (1, 10) =>1,2,3,4,5,6,7,8,9
• range (1, 10, 2) =>1,3,5,7,9
• range (10, 0, -1) =>10,9,8,7,6,5,4,3,2,1
• range (10, 0, -2) =>10,8,6,4,2
• range (2, 11, 2) =>2,4,6,8,10
• range (-5, 5) =>−5, −4, −3, −2, −1,0,1,2,3,4
• range (1, 2) =>1
• range (1, 1) =>(empty)
• range (1, -1) =>(empty)
• range (1, -1, -1) =>1,0
• range (0) =>(empty)

3. Nested Loops:

Just like with if statements, while and for blocks can contain arbitrary Python statements, including other loops. A loop can therefore be nested within another loop.

4. **The *break* statement:**

The *break* statement causes the immediate exit from the body of the loop. During a program's execution, when the *break* statement is encountered within the body of a loop, the remaining statements within the body of the loop are skipped, and the loop is exited.

```
for n in range (21, 0, -3):
        if n == 12:
                break
        print (n)
```

Output:
21
18
15

5. The *continue* statement:

When a *continue* statement is encountered within a loop, the remaining statements within the body are skipped, but the loop condition is checked to see if the loop should continue or be exited. If the loop's condition is still true, the loop is not exited, but the loop's execution continues at the top of the loop.

```
for n in range (21, 0, -3):
        if n == 12:
                continue
        print (n)
```

Output:
21
18
15
9
6
3

Modules

Python provides a collection of standard code stored in libraries called modules. Each module is a Python program that contains a related group of functions that can be embedded in your programs. For example, the math module has mathematics related functions, the random module has random number–related functions, and so on.

Before you can use the functions in a module, you must import the module with an import statement. In code, an import statement consists of the following:

i. The *import* keyword

ii. The name of the module

Method I

```
import random
for i in range (5):
        print (random. randint (1, 10))
```

In this method, whenever the functions from the module is to be used, the function name should followed by module name i.e

module.function()

Method II

```
from random import randint
for i in range (5):
        print (randint (1, 10))
```

In this method, whenever the functions from the module is to be used, only the function name is required i.e

function ()

Method III

```
from random import*
for i in range (5):
        print (randint (1, 10))
```

In this method, whenever the functions from the module is to be used, only the function name is required i.e

function ()

N.B: Method I and Method II imports all the function in that particular module. Method III ls only imports those functions which are mentioned.

Method I make the code lengthy because whenever the functions from the module is to be used, the function name should followed by module name whereas use of Method III is not a good practice because sometime it creates conflicts with keywords/objects.

It is advisable to use Method III

- We can import modules with alias name also e.g.

```
Import random as rd
for i in range (5):
        print (random. randint (1, 10))
```

Math Module:

Python has a module called math that contains familiar math functions.

ceil(x)	Rounds a number upwards to the nearest integer, and returns the result
cos(x)	Returns the cosine of x
cosh(x)	Returns the hyperbolic cosine of x
degrees(x)	Converts an angle from radians to degrees
fabs(x)	Returns the absolute value of a number
factorial (x)	Returns the factorial of x
floor(x)	Rounds a number downwards to the nearest integer, and returns the result
log (x)	Returns the natural logarithm of x
log10(x)	Returns the base-10 logarithm of x
pow (x, y)	Returns the value of x to the power of y
radians(x)	Converts a degree value (x) to radians
sin(x)	Returns the sine of x
sinh(x)	Returns the hyperbolic sine of x
sqrt(x)	Returns the square root of x
tan(x)	Returns the tangent of x
tanh(x)	Returns the hyperbolic tangent of x
acos(x)	Returns the arc cosine value of x
asin(x)	Returns the arc sine of x
atan(x)	Returns the arc tangent value of x
exp(x)	Returns the value of e^x
e	Returns Euler's number (2.7182...)
pi	Returns PI (3.1415...)

```
>>> x = complex (3.4, .3)
>>> x
(3.4+0.3j)
```

LISTS

Say we need to get thirty test scores from a user and do something with them, like put them in order. We could create thirty variables, score1, score2, . . . , score30, but that would be very tedious. To then put the scores in order would be extremely difficult. The solution is to use lists.

Creating lists Here is a simple list: L = [1,2,3]

Use square brackets to indicate the start and end of the list, and separate the items by commas.

The empty list: The empty list is []. It is the list equivalent of 0 or ''.

Printing lists You can use the **print** function to print the entire contents of a list.

```
>>>L = [1,2,3]
>>>print(L)
```

```
Output:
[1, 2, 3]
```

Data types: Lists can contain all kinds of things, even other lists. For example, the following is a valid list: [1, 2.718, 'abc', [5,6,7]]

Similarities to strings

There are a number of things which work the same way for lists as for strings.

• **len**— The number of items in L is given by **len**(L).

• **in**— The **in** operator tells you if a list contains something. Here are some examples:

```
if 2 in L:
        print ('Your list contains the number 2.')
if 0 not in L:
        print ('Your list has no zeroes.')
```

• **Indexing and slicing** — These work exactly as with strings. For example, L[0] is the first item of the list L and L [:3] gives the first three items.

The integer value -1 refers to the last index in a list, the value -2 refers to the second-to-last index in a list, and so on.

In a slice, the first integer is the index where the slice starts. The second integer is the index where the slice ends. A slice goes up to, *but will not include, the value at the second index*. A slice evaluates to a new list value.

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[0:4]
['cat', 'bat', 'rat', 'elephant']
>>> spam[1:3]
```

['bat', 'rat']

>>> **spam[0:-1]**

['cat', 'bat', 'rat']

As a shortcut, you can leave out one or both of the indexes on either side of the colon in the slice. Leaving out the first index is the same as using 0, or the beginning of the list. Leaving out the second index is the same as using the length of the list, which will slice to the end of the list.

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[:2]
['cat', 'bat']
>>> spam[1:]
['bat', 'rat', 'elephant']
>>> spam[:]
['cat', 'bat', 'rat', 'elephant']
```

• **+ and *** — The + operator adds one list to the end of another. The * operator repeats a list. Here are some examples:

Expression	Result
[7,8]+[3,4,5]	[7,8,3,4,5]
[7,8]*3	[7,8,7,8,7,8]
[0]*5	[0,0,0,0,0]

• **Looping:** The same two types of loops that work for strings also work for lists. Both of the following examples print out the items of a list, one-by-one, on separate lines.

```
for i in range(len(L)):          for item in L:
    print(L[i])                      print(item)
```

The left loop is useful for problems where you need to use the loop variable i to keep track of where you are in the loop. If that is not needed, then use the right loop, as it is a little simpler.

Built-in functions: There are several built-in functions that operate on lists. Here are some useful ones:

Function	Description
len	returns the number of items in the list
sum	returns the sum of the items in the list
min	returns the minimum of the items in the list
max	returns the maximum of the items in the list

For example, the following computes the average of the values in a list L:

average = **sum**(L)/**len**(L)

List methods: Here are some list methods:

Method	Description
append(x)	adds x to the end of the list
sort()	sorts the list
count(x)	returns the number of times x occurs in the list
index(x)	returns the location of the first occurrence of x
reverse()	reverses the list
remove(x)	removes first occurrence of x from the list
pop(p)	removes the item at index p and returns its value
insert(p,x)	inserts x at index p of the list

Miscellaneous

Making copies of lists: We have a list L and we want to make a copy of the list and call it M. The expression M=L will not work. For now, do the following in place of M=L:

```
M = L[:]
```

Changing lists: To change the value in location 2 of L to 100, we simply say L[2]=100. If we want to insert the value 100 into location 2 without overwriting what is currently there, we can use the insert method. To delete an entry from a list, we can use the **del** operator. Some examples are shown below. Assume L=[6,7,8] foreach operation.

Operation	New L	Description
L[1]= 9	[6,9,8]	replace item at index 1 with 9
L.insert(1,9)	[6,9,7,8]	insert a 9 at index 1 without replacing
del L[1]	[6,8]	delete second item
del L[:2]	[8]	delete first two items

User defined functions

The general form of a function definition is

def name (parameter list):

block

return (expression)

```
>>>def f(x):
        return x*x
>>> f(3)
>>> 9
```

```
>>> def greet ():
        print('Hello')
>>> greet ()
Hello
```

```
>>> def greet():
        print('Hello')
        return
>>> greet()
Hello
```

• The reserved word *def* signifies the beginning of a function definition.

• The name of the function is an identifier.

• The parameter list is a comma separated list of names that represent formal parameters to the function.

• The body is a block of statements. The statements define the actions that the function is to perform. The statements may include variables other than the function's formal parameters; unless specified otherwise, variables used with the function are ***local*** to that function.

• The *return* statement exits a function, optionally passing back an expression to the caller. A *return* statement with no arguments is the same as return none.

Recursion:

Recursion is the process where a function calls itself.

```
>>>def fact(n):
        if n==0:
                return 1
        else:
                return n*fact(n-1)
>>> fact(4)
24
>>>
```