

Operators and Expressions in Python

Basic Arithmetic operators in Python

Command	Name	Example	Output
+	Addition	4 + 5	9
-	Subtraction	8 - 5	3
*	Multiplication	4 * 5	20
/	True Division	19 / 3	6.3333
//	Integer Division	19//3	6
%	Remainder (modulo)	19 % 3	1
**	Exponent	2 ** 4	16

Order of Operations

Remember that thing called [order of operations](#) that they taught in maths? Well, it applies in Python, too. Here it is, if you need reminding:

1.parentheses ()

2.exponents **

3.multiplication *, division \, and remainder %

4.addition + and subtraction -

Order of Operations

Operator	Operation	Precedence
()	parentheses	0
**	exponentiation	1
*	multiplication	2
/	division	2
//	int division	2
%	remainder	2
+	addition	3
-	subtraction	3

- The computer scans the expression from left to right,
- first clearing parentheses,
- second, evaluating exponentiations from left to right in the order they are encountered
- third, evaluating $*$, $/$, $//$, $%$ from left to right in the order they are encountered,
- fourth, evaluating $+$, $-$ from left to right in the order they are encountered

$$2^{**}3+2^{*}(2+3)$$

- 18

Example 1 – Order of operations

```
>>> 1 + 2 * 3
```

7

```
>>> (1 + 2) * 3
```

9

- In the first example, the computer calculates $2 * 3$ first, then adds 1 to it. This is because multiplication has the higher priority (at 3) and addition is below that (at a lowly 4).
- In the second example, the computer calculates $1 + 2$ first, then multiplies it by 3. This is because parentheses have the higher priority (at 1), and addition comes in later than that.

Example 2 – Order of operations

Also remember that the math is calculated from left to right, *unless* you put in parentheses. The innermost parentheses are calculated first.

Watch these examples:

```
>>> 4 - 40 - 3
```

```
-39
```

```
>>> 4 - (40 - 3)
```

```
-33
```

- In the first example, $4 - 40$ is calculated, then $- 3$ is done.
- In the second example, $40 - 3$ is calculated, then it is subtracted from 4.

LET'S TRY IT

From the Python Shell, enter the following and observe the results.

```
>>> num = input('Enter number: ')
```

```
Enter number: 5
```

```
???
```

```
>>> num = int(input('Enter number: '))
```

```
Enter number: 5
```

```
???
```

```
>>> num = input('Enter name: ')
```

```
Enter name: John
```

```
???
```

```
>>> num = int(input('Enter name: '))
```

```
Enter name: John
```

```
???
```

Quotation in Python

- Python accepts single ('), double (") and triple (''' or ''') quotes to denote string literals, as long as the same type of quote starts and ends the string.
- The triple quotes are used to span the string across multiple lines. For example, all the following are legal –
- `word = 'word'`
- `sentence = "This is a sentence."`
- `paragraph = """This is a paragraph. It is made up of multiple lines and sentences."""`

Built-in format Function

- Because floating-point values may contain an arbitrary number of decimal places, the built-in **format** function can be used to produce a numeric string version of the value containing a specific number of decimal places.

```
>>> 12/5
```

```
2.4
```

```
>>> format(12/5, '.2f')
```

```
'2.40'
```

```
>>> 5/7
```

```
0.7142857142857143
```

```
>>> format(5/7, '.2f')
```

```
'0.71'
```

- In these examples, *format specifier* `'.2f'` rounds the result to two decimal places of accuracy in the string produced.

- For very large (or very small) values 'e' can be used as a format specifier,

```
>>> format(2 ** 100, '.6e')  
'1.267651e+30'
```

LET'S TRY IT

From the Python Shell, enter the following and observe the results.

```
>>> format(11/12, '.2f')
```

```
???
```

```
>>> format(11/12, '.2e')
```

```
???
```

```
>>> format(11/12, '.3f')
```

```
???
```

```
>>> format(11/12, '.3e')
```

```
???
```

Python is a Dynamic Type language

- Same variable can be associated with values of different type during program execution, as indicated below.
- It's also very dynamic as it rarely uses what it knows to limit variable usage

<code>var = 12</code>	integer
<code>var = 12.45</code>	float
<code>var = 'Hello'</code>	string

LET'S TRY IT

From the Python Shell, enter the following and observe the results.

```
>>> num = 10
```

```
>>> num
```

```
???
```

```
>>> id(num)
```

```
???
```

```
>>> num = 20
```

```
>>> num
```

```
???
```

```
>>> id(num)
```

```
???
```

```
>>> k = num
```

```
>>> k
```

```
???
```

```
>>> id(k)
```

```
???
```

```
>>> id(num)
```

```
???
```

```
>>> k = 30
```

```
>>> k
```

```
???
```

```
>>> num
```

```
???
```

```
>>> id(k)
```

```
???
```

```
>>> id(num)
```

```
???
```

```
>>> k = k + 1
```

```
>>> k
```

```
???
```

```
>>> id(num)
```

```
???
```

```
>>> id(k)
```

```
???
```

Bitwise Operations

- This includes operators that treat integers as strings of binary bits, and can come in handy if your Python code must deal with things like network packets, serial ports, or packed binary data
- ```
>>> x = 1
```

```
1 decimal is 0001 in bits
```

```
>>> x << 2
```

```
Shift left 2 bits: 0100
```
- 4



- $\ggg x \mid 2$       # Bitwise OR (either bit=1): 0011
- 3
- $\ggg x \& 1$       # Bitwise AND (both bits=1): 0001  
1
- In the first expression, a binary 1 (in base 2, 0001) is shifted left two slots to create a binary 4 (0100).
- The last two operations perform a binary OR to combine bits ( $0001 \mid 0010 = 0011$ ) and a binary AND to select common bits ( $0001 \& 0001 = 0001$ ).

- To print in binary format use bin function:
- `>>> X = 0b0001`      # Binary literals
- `>>> X << 2`      # Shift left 4
- `>>> bin(X << 2)`      # Binary digits string  
'0b100'
- `>>> bin(X | 0b010)`      # Bitwise OR: either  
'0b11'
- `>>> bin(X & 0b1)`      # Bitwise AND: both  
'0b0'

# Logical Operators

Assume a = 10 and b = 20

| Operator | Description                                                          | Example                |
|----------|----------------------------------------------------------------------|------------------------|
| and      | If both the operands are true then condition becomes true.           | (a and b) is true.     |
| Or       | If any of the two operands are non-zero then condition becomes true. | (a or b) is true.      |
| not      | Used to reverse the logical state of its operand.                    | Not(a and b) is false. |

# Python is a Strongly Typed language

- interpreter keeps track of all variable types
- Check type compatibility while expressions are evaluated
- `>>> 2+3`        `# right`
- `>>> "two"+1`    `# Wrong!!`

*Table 11-2. Augmented assignment statements*

$X += Y$      $X \&= Y$      $X -= Y$      $X |= Y$

$X *= Y$      $X ^= Y$      $X /= Y$      $X >>= Y$

$X \%= Y$      $X <<= Y$      $X **= Y$      $X //= Y$

---

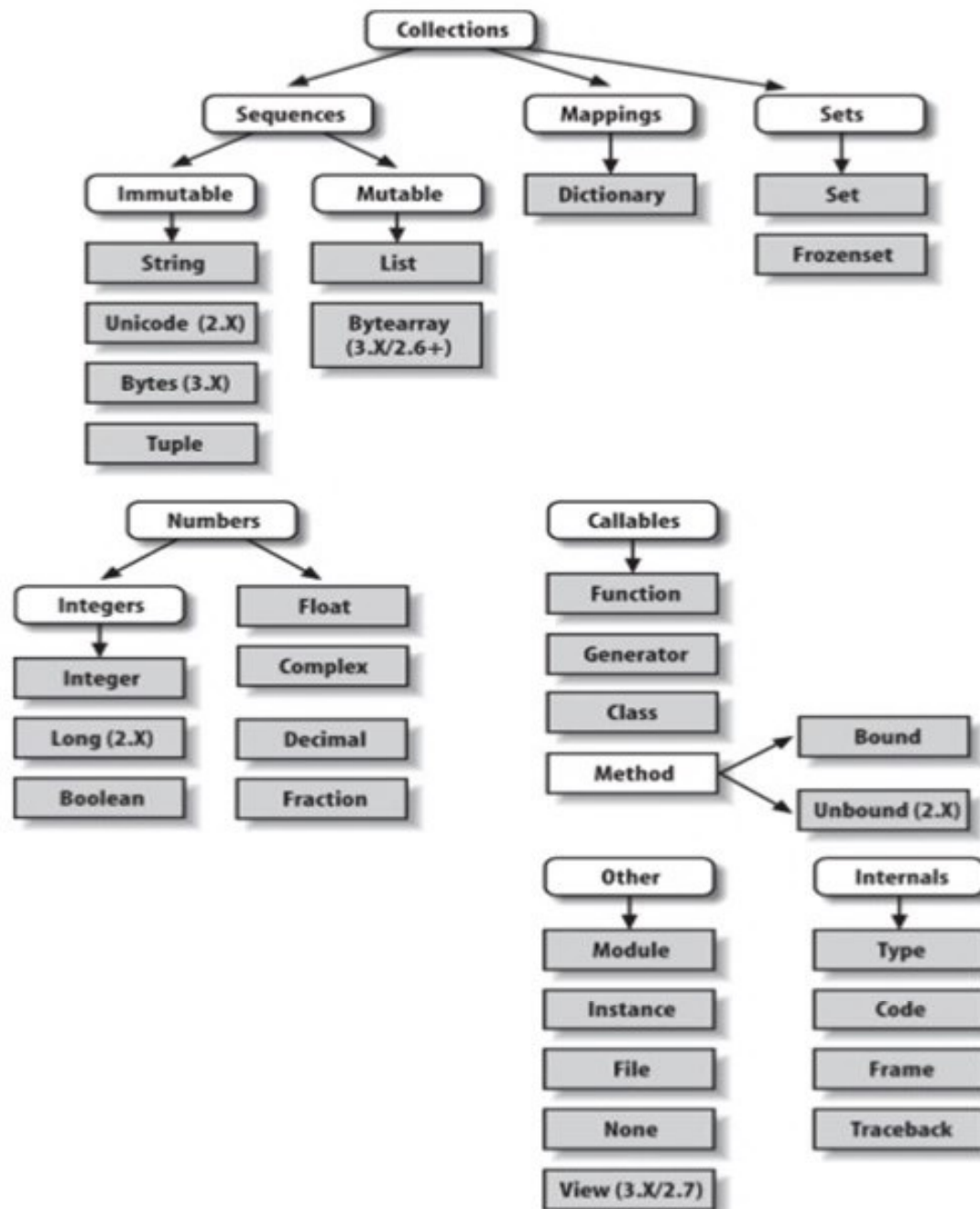


Figure 9-3. Python's major built-in object types, organized by categories. Everything is a type of object in Python, even the type of an object! Some extension types, such as named tuples, might belong in this figure too, but the criteria for inclusion in the core types set are not formal.

# Python Program for Bob Problem

```
n = float(input("Enter amount in hand"))
c = float(input("Enter price of one chocolate"))
m = int(input("Enter num of wrapper for free chocolate"))
#compute number of chocolates bought
p = n//c
#compute number of free chocolates
f = p//m
print("Number of chocolates",int(p+f))
```



# Problem -1

ABC company Ltd. is interested to computerize the pay calculation of their employee in the form of Basic Pay, Dearness Allowance (DA) and House Rent Allowance (HRA). DA and HRA are calculated as certain % of Basic pay(For example, DA is 80% of Basic Pay, and HRA is 30% of Basic pay). They have the deduction in the salary as PF which is 12% of Basic pay. Propose a computerized solution for the above said problem.

Input : Basic Pay

Process : Calculate Salary

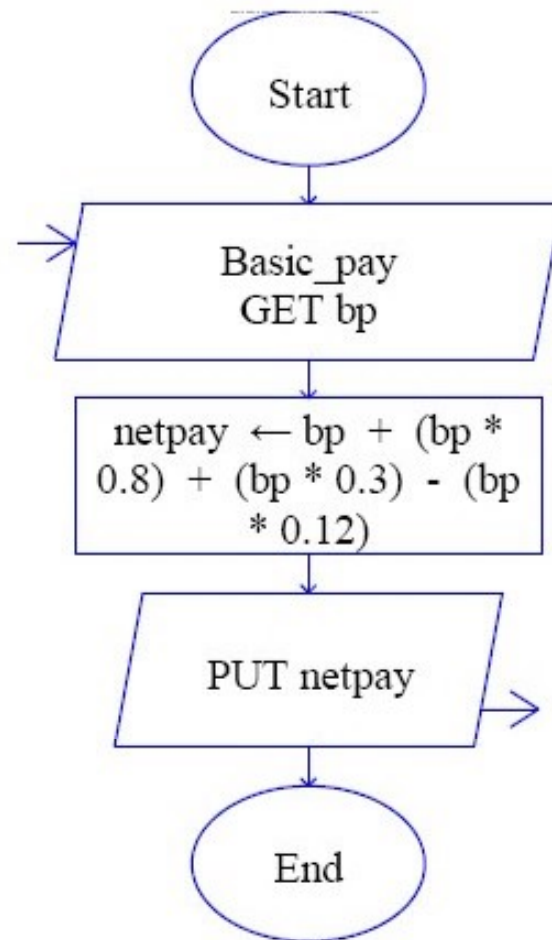
( Basic Pay + ( Basic Pay \* 0.8) + ( Basic Pay \* 0.3 - ( Basic Pay \* 0.12)

-----allowances ----- --- deductions----

Output : Salary



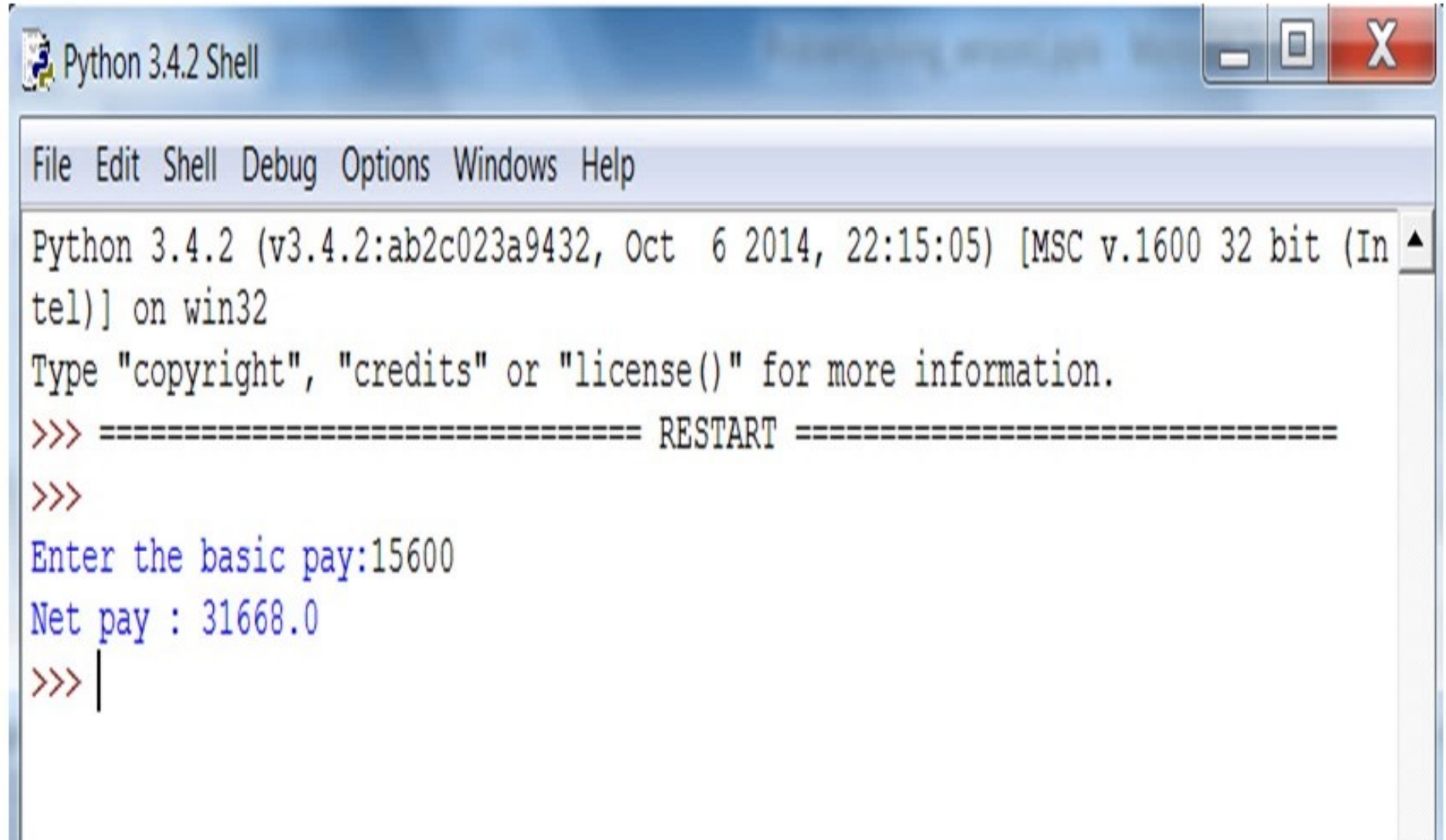
# Flow chart



# Python code

```
#Enter the basic pay
bp=float (input('Enter the basic pay:'))
net pay calucluation
netpay =bp + (bp*0.8) + (bp*0.3) - (bp*0.12)
display net salary
print ('Net pay :',netpay)
```

# Output



The image shows a screenshot of a Python 3.4.2 Shell window. The window has a title bar with the text 'Python 3.4.2 Shell' and standard Windows window controls (minimize, maximize, close). Below the title bar is a menu bar with the following items: File, Edit, Shell, Debug, Options, Windows, and Help. The main text area contains the following output:

```
Python 3.4.2 (v3.4.2:ab2c023a9432, Oct 6 2014, 22:15:05) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
Enter the basic pay:15600
Net pay : 31668.0
>>> |
```

# **Python features.....**

## **Lambda operator/function**

The lambda operator or lambda function is a way to create small anonymous functions, i.e. functions without a name.

```
>>> ftoc = lambda f: (f-32)*5.0/9
>>> ftoc(104)
```

# Python features.....

## Built-in Functions

The lambda operator or lambda function is a way to create small anonymous functions, i.e. functions without a name.

```
>>> ftoc = lambda f: (f-32)*5.0/9
>>> ftoc(104)
```

# Python features.....

## Built-in Functions in Python

| Built-in Functions                   |                                    |                                     |                                     |                                       |
|--------------------------------------|------------------------------------|-------------------------------------|-------------------------------------|---------------------------------------|
| <a href="#"><u>abs()</u></a>         | <a href="#"><u>delattr()</u></a>   | <a href="#"><u>hash()</u></a>       | <a href="#"><u>memoryview()</u></a> | <a href="#"><u>set()</u></a>          |
| <a href="#"><u>all()</u></a>         | <a href="#"><u>dict()</u></a>      | <a href="#"><u>help()</u></a>       | <a href="#"><u>min()</u></a>        | <a href="#"><u>setattr()</u></a>      |
| <a href="#"><u>any()</u></a>         | <a href="#"><u>dir()</u></a>       | <a href="#"><u>hex()</u></a>        | <a href="#"><u>next()</u></a>       | <a href="#"><u>slice()</u></a>        |
| <a href="#"><u>ascii()</u></a>       | <a href="#"><u>divmod()</u></a>    | <a href="#"><u>id()</u></a>         | <a href="#"><u>object()</u></a>     | <a href="#"><u>sorted()</u></a>       |
| <a href="#"><u>bin()</u></a>         | <a href="#"><u>enumerate()</u></a> | <a href="#"><u>input()</u></a>      | <a href="#"><u>oct()</u></a>        | <a href="#"><u>staticmethod()</u></a> |
| <a href="#"><u>bool()</u></a>        | <a href="#"><u>eval()</u></a>      | <a href="#"><u>int()</u></a>        | <a href="#"><u>open()</u></a>       | <a href="#"><u>str()</u></a>          |
| <a href="#"><u>breakpoint()</u></a>  | <a href="#"><u>exec()</u></a>      | <a href="#"><u>isinstance()</u></a> | <a href="#"><u>ord()</u></a>        | <a href="#"><u>sum()</u></a>          |
| <a href="#"><u>bytearray()</u></a>   | <a href="#"><u>filter()</u></a>    | <a href="#"><u>issubclass()</u></a> | <a href="#"><u>pow()</u></a>        | <a href="#"><u>super()</u></a>        |
| <a href="#"><u>bytes()</u></a>       | <a href="#"><u>float()</u></a>     | <a href="#"><u>iter()</u></a>       | <a href="#"><u>print()</u></a>      | <a href="#"><u>tuple()</u></a>        |
| <a href="#"><u>callable()</u></a>    | <a href="#"><u>format()</u></a>    | <a href="#"><u>len()</u></a>        | <a href="#"><u>property()</u></a>   | <a href="#"><u>type()</u></a>         |
| <a href="#"><u>chr()</u></a>         | <a href="#"><u>frozenset()</u></a> | <a href="#"><u>list()</u></a>       | <a href="#"><u>range()</u></a>      | <a href="#"><u>vars()</u></a>         |
| <a href="#"><u>classmethod()</u></a> | <a href="#"><u>getattr()</u></a>   | <a href="#"><u>locals()</u></a>     | <a href="#"><u>repr()</u></a>       | <a href="#"><u>zip()</u></a>          |
| <a href="#"><u>compile()</u></a>     | <a href="#"><u>globals()</u></a>   | <a href="#"><u>map()</u></a>        | <a href="#"><u>reversed()</u></a>   | <a href="#"><u>__import__()</u></a>   |
| <a href="#"><u>complex()</u></a>     | <a href="#"><u>hasattr()</u></a>   | <a href="#"><u>max()</u></a>        | <a href="#"><u>round()</u></a>      |                                       |

# Importing from Packages

- Packages are an essential building block in programming. Without packages, we'd spend lots of time writing code that's already been written.
- Imagine having to write code from scratch every time you wanted to parse a file in a particular format. You'd never get anything done! That's why we always want to use packages.



# Importing from Packages

- A Python package usually consists of several modules.
- Physically, a package is a folder containing modules and maybe other folders that themselves may contain more folders and modules.
- Conceptually, it's a namespace. This simply means that a package's modules are bound together by a package name, by which they may be referenced.



# Importing from Packages

- import a package using the import statement:  
`import <some_package>`
- Let's assume that we haven't yet installed any packages.
- Python comes with a big collection of pre-installed packages known as the Python Standard Library.
- It includes tools for a range of use cases, such as text processing and doing math.

## **Namespaces and Aliasing:**

- When we had imported the math module, we initialized the math namespace. This means that we can now refer to functions and classes from the math module by way of “dot notation”:

# Importing from Packages

- **import a package using the import statement:**

**# Example-1 : Square root calculation**

```
import math
math.sqrt(4)
```

**#Example-2 : using pi**

```
import math
```

```
print('pi is', math.pi)
print('cos(pi) is', math.cos(math.pi))
```

**# Example-3 : Aliasing**

```
Import math as m
m.factorial(5)
```

# Importing from Packages

- import a package using the import statement:

## Different ways of library calls: (Syntax Examples)

```
from math import sin, pi
import math
import math as m
from math import *
```

## Print commands:

```
print("sin(pi/2) =", sin(pi/2))
print("sin(pi/2) =", m.sin(m.pi/2))
print("sin(pi/2) =", math.sin(math.pi/2))
```

## List of Functions in Python “Math” Module

| Function                    | Description                                                       |
|-----------------------------|-------------------------------------------------------------------|
| <code>ceil(x)</code>        | Returns the smallest integer greater than or equal to x.          |
| <code>copysign(x, y)</code> | Returns x with the sign of y                                      |
| <code>fabs(x)</code>        | Returns the absolute value of x                                   |
| <code>factorial(x)</code>   | Returns the factorial of x                                        |
| <code>floor(x)</code>       | Returns the largest integer less than or equal to x               |
| <code>fmod(x, y)</code>     | Returns the remainder when x is divided by y                      |
| <code>frexp(x)</code>       | Returns the mantissa and exponent of x as the pair (m, e)         |
| <code>fsum(iterable)</code> | Returns an accurate floating point sum of values in the iterable  |
| <code>isfinite(x)</code>    | Returns True if x is neither an infinity nor a NaN (Not a Number) |
| <code>radians(x)</code>     | Converts angle x from degrees to radians                          |



|             |                                                                   |
|-------------|-------------------------------------------------------------------|
| isfinite(x) | Returns True if x is neither an infinity nor a NaN (Not a Number) |
| isinf(x)    | Returns True if x is a positive or negative infinity              |
| isnan(x)    | Returns True if x is a NaN                                        |
| ldexp(x, i) | Returns $x * (2^{**i})$                                           |
| modf(x)     | Returns the fractional and integer parts of x                     |
| trunc(x)    | Returns the truncated integer value of x                          |
| exp(x)      | Returns $e^{**x}$                                                 |
| expm1(x)    | Returns $e^{**x} - 1$                                             |
| log(x[, b]) | Returns the logarithm of x to the base b (defaults to e)          |
| log1p(x)    | Returns the natural logarithm of 1+x                              |
| log2(x)     | Returns the base-2 logarithm of x                                 |
| log10(x)    | Returns the base-10 logarithm of x                                |
| pow(x, y)   | Returns x raised to the power y                                   |
| sqrt(x)     | Returns the square root of x                                      |
| acos(x)     | Returns the arc cosine of x                                       |
| asin(x)     | Returns the arc sine of x                                         |
| atan(x)     | Returns the arc tangent of x                                      |