**Subject:** Python Programming

## Why Python?

1. Simple syntax
2. Length of code is very small
3. Complex problems can be solved in a simple manner.

## Features of Python:

1. Simple to learn
2. It is a open source
3. Portability
4. High level interpreter-automatic memory management
5. Object oriented programming
6. Supports nearly 200+ standard libraries.

## Applications of Python:

1. Web & Internet development
2. Desktop GUI Applications
3. Artificial Intelligence
4. Machine Learning
5. Image processing Applications
6. Games and 3D graphics
7. Network programming
8. Data base access
9. Business Applications

## Python Environments:

1. Anaconda ----- www.anaconda.com
2. IDLE-priginal---- www.python.org
3. Jupyter
4. Spyder
5. Pycharm
6. eRic
7. Th
8. Atom

## History of Python

1. Python is created by Dutch Guido van Rossum around 1991.
2. Python is an open-source project.
3. The mother site is **www.python.org**.

**Prepared by:** Suresh Kumar K

**Subject:** Python Programming

## Versions of Python:

1. Python 1: the initial version.
2. Python 2: released in 2000, with many new features such as garbage collector and support for Unicode.
3. Python 3 (Python 3000 or py3k): A major upgrade released in 2008. Python 3 is NOT backward compatible with Python 2.
4. To know your version of python type **Python --version**

## Python Statement:

1. Each line is treated as one statement in Python
2. No need to end a statement with semicolon
3. Multi line statements are also possible

## Python Syntax:

Python syntax can be executed by writing directly in the Command Line:

```
print(“Hello Python”)
```

C:/> python hello.py

## Python Comments:

- Comments can be used to explain Python code.
- Comments can be used to make the code more readable.
- Comments can be used to prevent execution when testing code.

## Creating a Comment:

Comments starts with a #, and Python will ignore them:

```
#This is a comment
print("Hello, World!")
```

# Multi Line Comments

Python does not really have a syntax for multi line comments.

To add a multiline comment you could insert a # for each line:

```
"""
This is a comment
written in
more than just one line
"""
print("Hello, World!")
```

As long as the string is not assigned to a variable, Python will read the code, but then ignore it, and you have made a multiline comment.

**Identifiers:**

1. Starting letter should be an alphabet or _
2. Other than _ , no other symbols are not allowed within the identifier.
3. Identifier name should not be keyword.
4. Length of an identifier is unlimited.
5. It is case sensitive.

Example: Name=”Suresh”

NAME=”suresh”

name=”suresh” these three are different variables.

**Variables:**

Variable is a container for storing data. Or named memory location.

There is no syntax for declaring variable in python.

Assigning value to a variable is itself both declaration and initialization.

Example: a=5

Example2:

```
x, y, z = "Orange", "Banana", "Cherry"
print(x)
print(y)
print(z)
```

Example3:

```
x = y = z = "Orange"
print(x)
```

```
print(y)
print(z)
```

Example4:

```
fruits = ["apple", "banana", "cherry"]
x, y, z = fruits
print(x)
print(y)
print(z)
```

Example5:

```
x = "Python"
y = "is"
z = "awesome"
print(x, y, z)
```

Example6:

```
x = "Python "
y = "is "
z = "awesome"
print(x + y + z)
```

Example7:

```
x = 5
y = 10
print(x + y)
```

## Data types in Python

Every value in Python has a datatype. Since everything is an object in Python programming, data types are actually classes and variables are instance (object) of these classes.

There are various data types in Python. Some of the important types are listed below.

## Python Numbers

Integers, floating point numbers and complex numbers fall under Python numbers category. They are defined as `int`, `float` and `complex` classes in Python.

We can use the `type()` function to know which class a variable or a value belongs to. Similarly, the `isinstance()` function is used to check if an object belongs to a particular class.

```
a = 5
print(a, "is of type", type(a))

a = 2.0
print(a, "is of type", type(a))

a = 1+2j
print(a, "is complex number?", isinstance(1+2j,complex))
```

**Output**

```
5 is of type <class 'int'>
2.0 is of type <class 'float'>
(1+2j) is complex number? True
```

Integers can be of any length, it is only limited by the memory available.

A floating-point number is accurate up to 15 decimal places. Integer and floating points are separated by decimal points. `1` is an integer, `1.0` is a floating-point number.
Complex numbers are written in the form, `x + yj`, where `x` is the real part and `y` is the imaginary part. Here are some examples.

```
>>> a = 1234567890123456789
>>> a
1234567890123456789
>>> b = 0.1234567890123456789
>>> b
0.12345678901234568
>>> c = 1+2j
>>> c
(1+2j)
```

Notice that the `float` variable `b` got truncated.

## Python List

List is an ordered sequence of items. It is one of the most used datatype in Python and is very flexible. All the items in a list do not need to be of the same type.

Declaring a list is pretty straight forward. Items separated by commas are enclosed within brackets `[ ]`.

```
a = [1, 2.2, 'python']
```

We can use the slicing operator `[ ]` to extract an item or a range of items from a list. The index starts from 0 in Python.

```
a = [5,10,15,20,25,30,35,40]

# a[2] = 15
print("a[2] = ", a[2])

# a[0:3] = [5, 10, 15]
print("a[0:3] = ", a[0:3])

# a[5:] = [30, 35, 40]
print("a[5:] = ", a[5:])
```

**Output**

```
a[2] =  15
a[0:3] =  [5, 10, 15]
a[5:] =  [30, 35, 40]
```

Lists are mutable, meaning, the value of elements of a list can be altered.

```
a = [1, 2, 3]
a[2] = 4
print(a)
```

**Output**

```
[1, 2, 4]
```

## Python Tuple

Tuple is an ordered sequence of items same as a list. The only difference is that tuples are immutable. Tuples once created cannot be modified. Tuples are used to write-protect data and are usually faster than lists as they cannot change dynamically.

It is defined within parentheses `()` where items are separated by commas.

```
t = (5,'program', 1+3j)
```

We can use the slicing operator `[]` to extract items but we cannot change its value.

```
t = (5,'program', 1+3j)

# t[1] = 'program'
print("t[1] = ", t[1])

# t[0:3] = (5, 'program', (1+3j))
print("t[0:3] = ", t[0:3])

# Generates error
# Tuples are immutable
t[0] = 10
```

**Output**

```
t[1] =  program
t[0:3] =  (5, 'program', (1+3j))
Traceback (most recent call last):
  File "test.py", line 11, in <module>
    t[0] = 10
TypeError: 'tuple' object does not support item assignment
```

# Python Strings

String is sequence of Unicode characters. We can use single quotes or double quotes to represent strings. Multi-line strings can be denoted using triple quotes, `'''` or `"""`.

```
s = "This is a string"
print(s)
s = '''A multiline
string'''
print(s)
```

**Output**

```
This is a string
A multiline
string
```

Just like a list and tuple, the slicing operator `[ ]` can be used with strings. Strings, however, are immutable.

```
s = 'Hello world!'

# s[4] = 'o'
print("s[4] = ", s[4])

# s[6:11] = 'world'
print("s[6:11] = ", s[6:11])

# Generates error
# Strings are immutable in Python
s[5] ='d'
```

**Output**

```
s[4] =  o
s[6:11] =  world
Traceback (most recent call last):
  File "<string>", line 11, in <module>
TypeError: 'str' object does not support item assignment
```

# Python Set

Set is an unordered collection of unique items. Set is defined by values separated by comma inside braces `{ }`. Items in a set are not ordered.

```
a = {5,2,3,1,4}

# printing set variable
print("a = ", a)

# data type of variable a
print(type(a))
```

**Output**

```
a =  {1, 2, 3, 4, 5}
<class 'set'>
```

We can perform set operations like union, intersection on two sets. Sets have unique values. They eliminate duplicates.

```
a = {1,2,2,3,3,3}
print(a)
```

**Output**

```
{1, 2, 3}
```

Since, set are unordered collection, indexing has no meaning. Hence, the slicing operator `[]` does not work.

```
>>> a = {1,2,3}
>>> a[1]
Traceback (most recent call last):
  File "<string>", line 301, in runcode
  File "<interactive input>", line 1, in <module>
TypeError: 'set' object does not support indexing
```

# Python Dictionary

Dictionary is an unordered collection of key-value pairs.
It is generally used when we have a huge amount of data. Dictionaries are optimized for retrieving data. We must know the key to retrieve the value.

In Python, dictionaries are defined within braces `{}` with each item being a pair in the form `key:value`. Key and value can be of any type.

```
>>> d = {1:'value','key':2}
>>> type(d)
<class 'dict'>
```

We use key to retrieve the respective value. But not the other way around.

```
d = {1:'value','key':2}
print(type(d))

print("d[1] = ", d[1])

print("d['key'] = ", d['key'])

# Generates error
print("d[2] = ", d[2])
```

**Output**

```
<class 'dict'>
d[1] =  value
d['key'] =  2
Traceback (most recent call last):
  File "<string>", line 9, in <module>
KeyError: 2
```

## Boolean

Boolean type provides two built-in values, True and False. These values are used to determine the given statement true or false. It denotes by the class bool. True can be represented by any non-zero value or 'T' whereas false can be represented by the 0 or 'F'. Consider the following example.

```
# Python program to check the boolean type

    print(type(True))
    print(type(False))
    print(false)
```

**Output:**

```
<class 'bool'>
<class 'bool'>
NameError: name 'false' is not defined
```

## Conversion between data types

We can convert between different data types by using different type conversion functions like `int()`, `float()`, `str()`, etc.

```
>>> float(5)
5.0
```

Conversion from float to int will truncate the value (make it closer to zero).

```
>>> int(10.6)
10
>>> int(-10.6)
-10
```

Conversion to and from string must contain compatible values.

```
>>> float('2.5')
2.5
>>> str(25)
```

```
'25'
>>> int('1p')
Traceback (most recent call last):
  File "<string>", line 301, in runcode
  File "<interactive input>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '1p'
```

We can even convert one sequence to another.

```
>>> set([1,2,3])
{1, 2, 3}
>>> tuple({5,6,7})
(5, 6, 7)
>>> list('hello')
['h', 'e', 'l', 'l', 'o']
```

To convert to dictionary, each element must be a pair:

```
>>> dict([[1,2],[3,4]])
{1: 2, 3: 4}
>>> dict([(3,26),(4,44)])
{3: 26, 4: 44}
```

## What are operators in python?

Operators are special symbols in Python that carry out arithmetic or logical computation. The value that the operator operates on is called the operand.

For example:

```
>>> 2+3
5
```

Here, `+` is the operator that performs addition. `2` and `3` are the operands and `5` is the output of the operation.

# Arithmetic operators

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication, etc.

| Operator | Meaning | Example |
|---|---|---|
| + | Add two operands or unary plus | x + y+ 2 |
| - | Subtract right operand from the left or unary minus | x - y- 2 |
| * | Multiply two operands | x * y |
| / | Divide left operand by the right one (always results into float) | x / y |
| % | Modulus - remainder of the division of left operand by the right | x % y (remainder of x/y) |
| // | Floor division - division that results into whole number adjusted to the left in the number line | x // y |
| ** | Exponent - left operand raised to the power of right | x**y (x to the power y) |

## Example 1: Arithmetic operators in Python

```
x = 15
y = 4

# Output: x + y = 19
print('x + y =',x+y)

# Output: x - y = 11
print('x - y =',x-y)

# Output: x * y = 60
print('x * y =',x*y)
```

```
# Output: x / y = 3.75
print('x / y =',x/y)

# Output: x // y = 3
print('x // y =',x//y)

# Output: x ** y = 50625
print('x ** y =',x**y)
```

## Output

```
x + y = 19
x - y = 11
x * y = 60
x / y = 3.75
x // y = 3
x ** y = 50625
```

## Comparison operators

Comparison operators are used to compare values. It returns either `True` or `False` according to the condition.

| Operator | Meaning | Example |
|---|---|---|
| > | Greater than - True if left operand is greater than the right | x > y |
| < | Less than - True if left operand is less than the right | x < y |
| == | Equal to - True if both operands are equal | x == y |
| != | Not equal to - True if operands are not equal | x != y |
| >= | Greater than or equal to - True if left operand is greater than or equal to the right | x >= y |
| <= | Less than or equal to - True if left operand is less than or equal to the right | x <= y |

### Example 2: Comparison operators in Python

```
x = 10
y = 12

# Output: x > y is False
print('x > y is',x>y)

# Output: x < y is True
print('x < y is',x<y)

# Output: x == y is False
print('x == y is',x==y)

# Output: x != y is True
print('x != y is',x!=y)

# Output: x >= y is False
print('x >= y is',x>=y)
```

```
# Output: x <= y is True
print('x <= y is',x<=y)
```

**Output**

```
x > y is False
x < y is True
x == y is False
x != y is True
x >= y is False
x <= y is True
```

## Logical operators

Logical operators are the `and`, `or`, `not` operators.

| Operator | Meaning | Example |
|---|---|---|
| and | True if both the operands are true | x and y |
| or | True if either of the operands is true | x or y |
| not | True if operand is false (complements the operand) | not x |

### Example 3: Logical Operators in Python

```
x = True
y = False

print('x and y is',x and y)

print('x or y is',x or y)

print('not x is',not x)
```

**Output**

```
x and y is False
x or y is True
not x is False
```

## Bitwise operators

Bitwise operators act on operands as if they were strings of binary digits. They operate bit by bit, hence the name.

For example, 2 is `10` in binary and 7 is `111`.
**In the table below:** Let `x` = 10 (`0000 1010` in binary) and `y` = 4 (`0000 0100` in binary)

| Operator | Meaning | Example |
|---|---|---|
| & | Bitwise AND | x & y = 0 (`0000 0000`) |
| \| | Bitwise OR | x \| y = 14 (`0000 1110`) |
| ~ | Bitwise NOT | ~x = -11 (`1111 0101`) |
| ^ | Bitwise XOR | x ^ y = 14 (`0000 1110`) |
| >> | Bitwise right shift | x >> 2 = 2 (`0000 0010`) |
| << | Bitwise left shift | x << 2 = 40 (`0010 1000`) |

## Assignment operators

Assignment operators are used in Python to assign values to variables.

`a = 5` is a simple assignment operator that assigns the value 5 on the right to the variable `a` on the left.
There are various compound operators in Python like `a += 5` that adds to the variable and later assigns the same. It is equivalent to `a = a + 5`.

| Operator | Example | Equivalent to |
|---|---|---|
| = | x = 5 | x = 5 |
| += | x += 5 | x = x + 5 |
| -= | x -= 5 | x = x - 5 |
| *= | x *= 5 | x = x * 5 |
| /= | x /= 5 | x = x / 5 |
| %= | x %= 5 | x = x % 5 |
| //= | x //= 5 | x = x // 5 |
| **= | x **= 5 | x = x ** 5 |
| &= | x &= 5 | x = x & 5 |
| \|= | x \|= 5 | x = x \| 5 |
| ^= | x ^= 5 | x = x ^ 5 |
| >>= | x >>= 5 | x = x >> 5 |
| <<= | x <<= 5 | x = x << 5 |

# Special operators

Python language offers some special types of operators like the identity operator or the membership operator. They are described below with examples.

## Identity operators

`is` and `is not` are the identity operators in Python. They are used to check if two values (or variables) are located on the same part of the memory. Two variables that are equal does not imply that they are identical.

| Operator | Meaning | Example |
|---|---|---|
| Is | True if the operands are identical (refer to the same object) | x is True |
| is not | True if the operands are not identical (do not refer to the same object) | x is not True |

## Example 4: Identity operators in Python

```
x1 = 5
y1 = 5
x2 = 'Hello'
y2 = 'Hello'
x3 = [1,2,3]
y3 = [1,2,3]

# Output: False
print(x1 is not y1)

# Output: True
print(x2 is y2)

# Output: False
print(x3 is y3)
```

### Output

```
False
True
False
```

Here, we see that `x1` and `y1` are integers of the same values, so they are equal as well as identical. Same is the case with `x2` and `y2` (strings).
But `x3` and `y3` are lists. They are equal but not identical. It is because the interpreter locates them separately in memory although they are equal.

## Membership operators

`in` and `not in` are the membership operators in Python. They are used to test whether a value or variable is found in a sequence (string, list, tuple, set and dictionary).
In a dictionary we can only test for presence of key, not the value.

| Operator | Meaning | Example |
|---|---|---|
| in | True if value/variable is found in the sequence | 5 in x |
| not in | True if value/variable is not found in the sequence | 5 not in x |

## Example #5: Membership operators in Python

```
x = 'Hello world'
y = {1:'a',2:'b'}

# Output: True
print('H' in x)

# Output: True
print('hello' not in x)

# Output: True
print(1 in y)
```

```
# Output: False
print('a' in y)
```

## Output

```
True
True
True
False
```

Here, `'H'` is in `x` but `'hello'` is not present in `x` (remember, Python is case sensitive). Similarly, `1` is key and `'a'` is the value in dictionary `y`. Hence, `'a' in y` returns `False`.

Strings can be *indexed* (subscripted), with the first character having index 0. There is no separate character type; a character is simply a string of size one:

```
>>> word = 'Python'
>>> word[0]  # character in position 0
'P'
>>> word[5]  # character in position 5
'n'
```

Indices may also be negative numbers, to start counting from the right:

```
>>> word[-1]  # last character
'n'
>>> word[-2]  # second-last character
'o'
>>> word[-6]
'P'
```

Note that since -0 is the same as 0, negative indices start from -1.

In addition to indexing, *slicing* is also supported. While indexing is used to obtain individual characters, *slicing* allows you to obtain substring:

```
>>> word[0:2]  # characters from position 0 (included) to 2 (excluded)
'Py'
```

```
>>> word[2:5]  # characters from position 2 (included) to 5
(excluded)
'tho'
```

Slice indices have useful defaults; an omitted first index defaults to zero, an omitted second index defaults to the size of the string being sliced.

```
>>>
```

```
>>> word[:2]   # character from the beginning to position 2
(excluded)
'Py'
>>> word[4:]   # characters from position 4 (included) to the end
'on'
>>> word[-2:]  # characters from the second-last (included) to
the end
'on'
```

Note how the start is always included, and the end always excluded. This makes sure that `s[:i] + s[i:]` is always equal to `s`:

```
>>>
```

```
>>> word[:2] + word[2:]
'Python'
>>> word[:4] + word[4:]
'Python'
```

One way to remember how slices work is to think of the indices as pointing *between* characters, with the left edge of the first character numbered 0. Then the right edge of the last character of a string of *n* characters has index *n*, for example:

```
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

The first row of numbers gives the position of the indices 0…6 in the string; the second row gives the corresponding negative indices. The slice from *i* to *j* consists of all characters between the edges labeled *i* and *j*, respectively.

For non-negative indices, the length of a slice is the difference of the indices, if both are within bounds. For example, the length of `word[1:3]` is 2.

Attempting to use an index that is too large will result in an error:

```
>>>
```

```
>>> word[42]  # the word only has 6 characters
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

However, out of range slice indexes are handled gracefully when used for slicing:

## What are lambda functions in Python?

In Python, an anonymous function is a function that is defined without a name.

While normal functions are defined using the `def` keyword in Python, anonymous functions are defined using the `lambda` keyword.

Hence, anonymous functions are also called lambda functions.

## How to use lambda Functions in Python?

A lambda function in python has the following syntax.

### Syntax of Lambda Function in python

```
lambda arguments: expression
```

Lambda functions can have any number of arguments but only one expression. The expression is evaluated and returned. Lambda functions can be used wherever function objects are required.

### Example of Lambda Function in python

Here is an example of lambda function that doubles the input value.

```
# Program to show the use of lambda functions
double = lambda x: x * 2

print(double(5))
```

**Output**

```
10
```

In the above program, `lambda x: x * 2` is the lambda function. Here `x` is the argument and `x * 2` is the expression that gets evaluated and returned.
This function has no name. It returns a function object which is assigned to the identifier `double`. We can now call it as a normal function. The statement

```
double = lambda x: x * 2
```

is nearly the same as:

```
def double(x):

   return x * 2
```

## Use of Lambda Function in python

We use lambda functions when we require a nameless function for a short period of time.

In Python, we generally use it as an argument to a higher-order function (a function that takes in other functions as arguments). Lambda functions are used along with built-in functions like `filter()`, `map()` etc.

### Example use with filter()

The `filter()` function in Python takes in a function and a list as arguments. The function is called with all the items in the list and a new list is returned which contains items for which the function evaluates to `True`.
Here is an example use of `filter()` function to filter out only even numbers from a list.

```
# Program to filter out only the even items from a list
my_list = [1, 5, 4, 6, 8, 11, 3, 12]

new_list = list(filter(lambda x: (x%2 == 0) , my_list))

print(new_list)
```

Run Code

**Output**

```
[4, 6, 8, 12]
```

### Example use with map()

The `map()` function in Python takes in a function and a list.

The function is called with all the items in the list and a new list is returned which contains items returned by that function for each item.

Here is an example use of `map()` function to double all the items in a list.

```
# Program to double each item in a list using map()

my_list = [1, 5, 4, 6, 8, 11, 3, 12]

new_list = list(map(lambda x: x * 2 , my_list))

print(new_list)
```

Run Code

**Output**

```
[2, 10, 8, 12, 16, 22, 6, 24]
```

# What is a Module?

Consider a module to be the same as a code library.

A file containing a set of functions you want to include in your application.

# Create a Module

To create a module just save the code you want in a file with the file extension `.py`:

## Example

Save this code in a file named `mymodule.py`

```
def greeting(name):
  print("Hello, " + name)
```

```
def sum1(a,b):
      c=a+b
      print(“sum is”,c)
```

# Use a Module

Now we can use the module we just created, by using the `import` statement:

## Example

Import the module named mymodule, and call the greeting function:

```
import mymodule

mymodule.greeting("Suresh")
```

Run Example »

**Note:** When using a function from a module, use the syntax: *module_name.function_name*.

# Variables in Module

The module can contain functions, as already described, but also variables of all types (arrays, dictionaries, objects etc):

## Example

Save this code in the file `mymodule.py`

```
person1 = {
  "name": "John",
  "age": 36,
  "country": "Norway"
}
```

## Example

Import the module named mymodule, and access the person1 dictionary:

```
import mymodule

a = mymodule.person1["age"]
print(a)
```

**O/P: 36**

# Naming a Module

You can name the module file whatever you like, but it must have the file extension `.py`

# Re-naming a Module

You can create an alias when you import a module, by using the `as` keyword:

## Example

Create an alias for `mymodule` called `mx`:

```
import mymodule as mx

a = mx.person1["age"]
print(a)
```

**O/P: 36**

# Built-in Modules

There are several built-in modules in Python, which you can import whenever you like.

## Example

Import and use the `platform` module:

```
import platform

x = platform.system()
print(x)
```

# Python datetime

Python has a module named **datetime** to work with dates and times.

## Example 1: Get Current Date and Time

```
import datetime

datetime_object = datetime.datetime.now()
print(datetime_object)
```

When you run the program, the output will be something like:

```
2018-12-19 09:26:03.478039
```

Here, we have imported **datetime** module using `import datetime` statement. One of the classes defined in the `datetime` module is `datetime` class. We then used `now()` method to create a `datetime` object containing the current local date and time.

## Example 2: Get Current Date

```
from datetime import date

date_object = date.today()
print(date_object)
```

When you run the program, the output will be something like:

```
2018-12-19
```

In this program, we have used `today()` method defined in the `date` class to get a `date` object containing the current local date.

**What's inside datetime?**

We can use dir() function to get a list containing all attributes of a module.

```
import datetime
```

```
print(dir(datetime))
```

When you run the program, the output will be:

```
['MAXYEAR', 'MINYEAR', '__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__', '_divide_and_round', 'date',
'datetime', 'datetime_CAPI', 'time', 'timedelta', 'timezone', 'tzinfo']
```

Commonly used classes in the datetime module are:

- date Class
- time Class
- datetime Class
- timedelta Class

## datetime.date Class

You can instantiate `date` objects from the `date` class. A date object represents a date (year, month and day).

### Example 3: Date object to represent a date

```
import datetime

d = datetime.date(2019, 4, 13)
print(d)
```

When you run the program, the output will be:

```
2019-04-13
```

If you are wondering, `date()` in the above example is a constructor of the `date` class. The constructor takes three arguments: year, month and day.

The variable `a` is a `date` object.

We can only import `date` class from the `datetime` module. Here's how:

```
from datetime import date

a = date(2019, 4, 13)
print(a)
```

## Example 4: Get current date

You can create a `date` object containing the current date by using a classmethod named `today()`. Here's how:

```
from datetime import date

today = date.today()

print("Current date =", today)
```

## Example 5: Get date from a timestamp

We can also create `date` objects from a timestamp. A Unix timestamp is the number of seconds between a particular date and January 1, 1970 at UTC. You can convert a timestamp to date using `fromtimestamp()` method.

```
from datetime import date

timestamp = date.fromtimestamp(1326244364)
print("Date =", timestamp)
```

When you run the program, the output will be:

```
Date = 2012-01-11
```

## Example 6: Print today's year, month and day

We can get year, month, day, day of the week etc. from the date object easily. Here's how:

```
from datetime import date

# date object of today's date
today = date.today()

print("Current year:", today.year)
print("Current month:", today.month)
print("Current day:", today.day)
```

## datetime.time

A time object instantiated from the `time` class represents the local time.

### Example 7: Time object to represent time

```
from datetime import time

# time(hour = 0, minute = 0, second = 0)
a = time()
print("a =", a)

# time(hour, minute and second)
b = time(11, 34, 56)
print("b =", b)

# time(hour, minute and second)
c = time(hour = 11, minute = 34, second = 56)
print("c =", c)

# time(hour, minute, second, microsecond)
d = time(11, 34, 56, 234566)
print("d =", d)
```

When you run the program, the output will be:

```
a = 00:00:00
b = 11:34:56
c = 11:34:56
d = 11:34:56.234566
```

### Example 8: Print hour, minute, second and microsecond

Once you create a `time` object, you can easily print its attributes such as `hour`, `minute` etc.

```
from datetime import time

a = time(11, 34, 56)

print("hour =", a.hour)
print("minute =", a.minute)
print("second =", a.second)
print("microsecond =", a.microsecond)
```

When you run the example, the output will be:

```
hour = 11
minute = 34
second = 56
microsecond = 0
```

Notice that we haven't passed `microsecond` argument. Hence, its default value `0` is printed.

## datetime.datetime

The `datetime` module has a class named **datetime** that can contain information from both **date** and **time** objects.

### Example 9: Python datetime object

```
from datetime import datetime

#datetime(year, month, day)
a = datetime(2018, 11, 28)
print(a)
```

```
# datetime(year, month, day, hour, minute, second, microsecond)
b = datetime(2017, 11, 28, 23, 55, 59, 342380)
print(b)
```

When you run the program, the output will be:

```
2018-11-28 00:00:00
2017-11-28 23:55:59.342380
```

The first three arguments `year`, `month` and `day` in the `datetime()` constructor are mandatory.

### Example 10: Print year, month, hour, minute and timestamp

```
from datetime import datetime

a = datetime(2017, 11, 28, 23, 55, 59, 342380)
print("year =", a.year)
print("month =", a.month)
print("hour =", a.hour)
print("minute =", a.minute)
print("timestamp =", a.timestamp())
```

When you run the program, the output will be:

```
year = 2017
month = 11
day = 28
hour = 23
minute = 55
timestamp = 1511913359.34238
```

## datetime.timedelta

A `timedelta` object represents the difference between two dates or times.

### Example 11: Difference between two dates and times

```
```

```
from datetime import datetime, date

t1 = date(year = 2018, month = 7, day = 12)
t2 = date(year = 2017, month = 12, day = 23)
t3 = t1 - t2
print("t3 =", t3)

t4 = datetime(year = 2018, month = 7, day = 12, hour = 7, minute = 9, second = 33)
t5 = datetime(year = 2019, month = 6, day = 10, hour = 5, minute = 55, second = 13)
t6 = t4 - t5
print("t6 =", t6)

print("type of t3 =", type(t3))
print("type of t6 =", type(t6))
```

When you run the program, the output will be:

```
t3 = 201 days, 0:00:00
t6 = -333 days, 1:14:20
type of t3 = <class 'datetime.timedelta'>
type of t6 = <class 'datetime.timedelta'>
```

Notice, both `t3` and `t6` are of `<class 'datetime.timedelta'>` type.

## Example 12: Difference between two timedelta objects

```
from datetime import timedelta

t1 = timedelta(weeks = 2, days = 5, hours = 1, seconds = 33)
t2 = timedelta(days = 4, hours = 11, minutes = 4, seconds = 54)
t3 = t1 - t2

print("t3 =", t3)
```

When you run the program, the output will be:

```
t3 = 14 days, 13:55:39
```

Here, we have created two `timedelta` objects `t1` and `t2`, and their difference is printed on the screen.

### Example 13: Printing negative timedelta object

```
from datetime import timedelta

t1 = timedelta(seconds = 33)
t2 = timedelta(seconds = 54)
t3 = t1 - t2

print("t3 =", t3)
print("t3 =", abs(t3))
```

When you run the program, the output will be:

```
t3 = -1 day, 23:59:39
t3 = 0:00:21
```

### Example 14: Time duration in seconds

You can get the total number of seconds in a timedelta object using `total_seconds()` method.

```
from datetime import timedelta

t = timedelta(days = 5, hours = 1, seconds = 33, microseconds = 233423)
print("total seconds =", t.total_seconds())
```

When you run the program, the output will be:

```
total seconds = 435633.233423
```

You can also find sum of two dates and times using `+` operator. Also, you can multiply and divide a `timedelta` object by integers and floats.

## Python format datetime

The way date and time is represented may be different in different places, organizations etc. It's more common to use `mm/dd/yyyy` in the US, whereas `dd/mm/yyyy` is more common in the UK.

Python has `strftime()` and `strptime()` methods to handle this.

## Python strftime() - datetime object to string

The `strftime()` method is defined under classes `date`, `datetime` and `time`. The method creates a formatted string from a given `date`, `datetime` or `time` object.

## Example 15: Format date using strftime()

```
from datetime import datetime

# current date and time
a = datetime.now()

t = a.strftime("%H:%M:%S")
print("time:", t)

s1 = a.strftime("%m/%d/%Y, %H:%M:%S")
# mm/dd/YY H:M:S format
print("s1:", s1)

s2 = a.strftime("%d/%m/%Y, %H:%M:%S")
# dd/mm/YY H:M:S format
print("s2:", s2)
```

When you run the program, the output will be something like:

```
time: 04:34:52
s1: 12/26/2018, 04:34:52
s2: 26/12/2018, 04:34:52
```

Here, `%Y`, `%m`, `%d`, `%H` etc. are format codes. The `strftime()` method takes one or more format codes and returns a formatted string based on it.

In the above program, `t`, `s1` and `s2` are strings.

- `%Y` - year [0001,..., 2018, 2019,..., 9999]
- `%m` - month [01, 02, ..., 11, 12]
- `%d` - day [01, 02, ..., 30, 31]
- `%H` - hour [00, 01, ..., 22, 23
- `%M` - minute [00, 01, ..., 58, 59]

- `%S` - second [00, 01, ..., 58, 59]

## Format Code List

The table below shows all the codes that you can pass to the `strftime()` method.

| Directive | Meaning | Example |
|---|---|---|
| `%a` | Abbreviated weekday name. | Sun, Mon, ... |
| `%A` | Full weekday name. | Sunday, Monday, ... |
| `%w` | Weekday as a decimal number. | 0, 1, ..., 6 |
| `%d` | Day of the month as a zero-padded decimal. | 01, 02, ..., 31 |
| `%-d` | Day of the month as a decimal number. | 1, 2, ..., 30 |
| `%b` | Abbreviated month name. | Jan, Feb, ..., Dec |
| `%B` | Full month name. | January, February, ... |
| `%m` | Month as a zero-padded decimal number. | 01, 02, ..., 12 |
| `%-m` | Month as a decimal number. | 1, 2, ..., 12 |
| `%y` | Year without century as a zero-padded decimal number. | 00, 01, ..., 99 |
| `%-y` | Year without century as a decimal number. | 0, 1, ..., 99 |
| `%Y` | Year with century as a decimal number. | 2013, 2019 |

| Directive | Meaning | Example |
|---|---|---|
| | | etc. |
| %H | Hour (24-hour clock) as a zero-padded decimal number. | 00, 01, ..., 23 |
| %-H | Hour (24-hour clock) as a decimal number. | 0, 1, ..., 23 |
| %I | Hour (12-hour clock) as a zero-padded decimal number. | 01, 02, ..., 12 |
| %-I | Hour (12-hour clock) as a decimal number. | 1, 2, ... 12 |
| %p | Locale's AM or PM. | AM, PM |
| %M | Minute as a zero-padded decimal number. | 00, 01, ..., 59 |
| %-M | Minute as a decimal number. | 0, 1, ..., 59 |
| %S | Second as a zero-padded decimal number. | 00, 01, ..., 59 |
| %-S | Second as a decimal number. | 0, 1, ..., 59 |
| %f | Microsecond as a decimal number, zero-padded on the left. | 000000 - 999999 |
| %z | UTC offset in the form +HHMM or -HHMM. | |
| %Z | Time zone name. | |
| %j | Day of the year as a zero-padded decimal number. | 001, 002, ..., 366 |
| %-j | Day of the year as a decimal number. | 1, 2, ..., 366 |

| Directive | Meaning | Example |
|---|---|---|
| %U | Week number of the year (Sunday as the first day of the week). All days in a new year preceding the first Sunday are considered to be in week 0. | 00, 01, ..., 53 |
| %W | Week number of the year (Monday as the first day of the week). All days in a new year preceding the first Monday are considered to be in week 0. | 00, 01, ..., 53 |
| %c | Locale's appropriate date and time representation. | Mon Sep 30 07:06:05 2013 |
| %x | Locale's appropriate date representation. | 09/30/13 |
| %X | Locale's appropriate time representation. | 07:06:05 |
| %% | A literal '%' character. | % |

# Files

Files are named locations on disk to store related information. They are used to permanently store data in a non-volatile memory (e.g. hard disk).

Since Random Access Memory (RAM) is volatile (which loses its data when the computer is turned off), we use files for future use of the data by permanently storing them.

When we want to read from or write to a file, we need to open it first. When we are done, it needs to be closed so that the resources that are tied with the file are freed.

Hence, in Python, a file operation takes place in the following order:

1. Open a file

2. Read or write (perform operation)
3. Close the file

## Opening Files in Python

Python has a built-in `open()` function to open a file. This function returns a file object, also called a handle, as it is used to read or modify the file accordingly.

```
>>> f = open("test.txt")    # open file in current directory
>>> f = open("C:/Python38/README.txt")  # specifying full path
```

We can specify the mode while opening a file. In mode, we specify whether we want to read `r`, write `w` or append `a` to the file. We can also specify if we want to open the file in text mode or binary mode.
The default is reading in text mode. In this mode, we get strings when reading from the file.

On the other hand, binary mode returns bytes and this is the mode to be used when dealing with non-text files like images or executable files.

| Mode | Description |
|---|---|
| `r` | Opens a file for reading. (default) |
| `w` | Opens a file for writing. Creates a new file if it does not exist or truncates the file if it exists. |
| `x` | Opens a file for exclusive creation. If the file already exists, the operation fails. |
| `a` | Opens a file for appending at the end of the file without truncating it. Creates a new file if it does not exist. |

| | |
|---|---|
| `t` | Opens in text mode. (default) |
| `b` | Opens in binary mode. |
| `+` | Opens a file for updating (reading and writing) |

```
f = open("test.txt")      # equivalent to 'r' or 'rt'
f = open("test.txt",'w')  # write in text mode
f = open("img.bmp",'r+b') # read and write in binary mode
```

Unlike other languages, the character `a` does not imply the number 97 until it is encoded using `ASCII` (or other equivalent encodings).
Moreover, the default encoding is platform dependent. In windows, it is `cp1252` but `utf-8` in Linux.
So, we must not also rely on the default encoding or else our code will behave differently in different platforms.

Hence, when working with files in text mode, it is highly recommended to specify the encoding type.

```
f = open("test.txt", mode='r', encoding='utf-8')
```

## Closing Files in Python

When we are done with performing operations on the file, we need to properly close the file.

Closing a file will free up the resources that were tied with the file. It is done using the `close()` method available in Python.
Python has a garbage collector to clean up unreferenced objects but we must not rely on it to close the file.

```
f = open("test.txt", encoding = 'utf-8')
# perform file operations
f.close()
```

This method is not entirely safe. If an exception occurs when we are performing some operation with the file, the code exits without closing the file.

A safer way is to use a try...finally block.

```
try:
   f = open("test.txt", encoding = 'utf-8')
   # perform file operations
finally:
   f.close()
```

This way, we are guaranteeing that the file is properly closed even if an exception is raised that causes program flow to stop.

The best way to close a file is by using the `with` statement. This ensures that the file is closed when the block inside the `with` statement is exited.
We don't need to explicitly call the `close()` method. It is done internally.

```
with open("test.txt", encoding = 'utf-8') as f:
   # perform file operations
```

## Writing to Files in Python

In order to write into a file in Python, we need to open it in write `w`, append `a` or exclusive creation `x` mode.
We need to be careful with the `w` mode, as it will overwrite into the file if it already exists. Due to this, all the previous data are erased.

Writing a string or sequence of bytes (for binary files) is done using the `write()` method. This method returns the number of characters written to the file.

```
with open("test.txt",'w',encoding = 'utf-8') as f:
   f.write("my first file\n")
   f.write("This file\n\n")
   f.write("contains three lines\n")
```

This program will create a new file named `test.txt` in the current directory if it does not exist. If it does exist, it is overwritten.
We must include the newline characters ourselves to distinguish the different lines.

## Reading Files in Python

To read a file in Python, we must open the file in reading `r` mode.
There are various methods available for this purpose. We can use the `read(size)` method to read in the `size` number of data. If the `size` parameter is not specified, it reads and returns up to the end of the file.
We can read the `text.txt` file we wrote in the above section in the following way:

```
>>> f = open("test.txt",'r',encoding = 'utf-8')
>>> f.read(4)    # read the first 4 data
'This'

>>> f.read(4)    # read the next 4 data
' is '

>>> f.read()     # read in the rest till end of file
'my first file\nThis file\ncontains three lines\n'

>>> f.read()  # further reading returns empty sting
```

```
''
```

We can see that the `read()` method returns a newline as `'\n'`. Once the end of the file is reached, we get an empty string on further reading.
We can change our current file cursor (position) using the `seek()` method. Similarly, the `tell()` method returns our current position (in number of bytes).

```
>>> f.tell()    # get the current file position
56

>>> f.seek(0)   # bring file cursor to initial position
0

>>> print(f.read())  # read the entire file
This is my first file
This file
contains three lines
```

We can read a file line-by-line using a for loop. This is both efficient and fast.

```
>>> for line in f:
...     print(line, end = '')
...
This is my first file
This file
contains three lines
```

In this program, the lines in the file itself include a newline character `\n`. So, we use the end parameter of the `print()` function to avoid two newlines when printing.
Alternatively, we can use the `readline()` method to read individual lines of a file. This method reads a file till the newline, including the newline character.

```
>>> f.readline()
'This is my first file\n'

>>> f.readline()
'This file\n'
```

```
>>> f.readline()
'contains three lines\n'

>>> f.readline()
''
```

Lastly, the `readlines()` method returns a list of remaining lines of the entire file. All these reading methods return empty values when the end of file (EOF) is reached.

```
>>> f.readlines()
['This is my first file\n', 'This file\n', 'contains three lines\n']
```

## Python File Methods

There are various methods available with the file object. Some of them have been used in the above examples.

Here is the complete list of methods in text mode with a brief description:

| Method | Description |
|---|---|
| close() | Closes an opened file. It has no effect if the file is already closed. |
| detach() | Separates the underlying binary buffer from the `TextIOBase` and returns it. |
| fileno() | Returns an integer number (file descriptor) of the file. |
| flush() | Flushes the write buffer of the file stream. |
| isatty() | Returns `True` if the file stream is interactive. |

| | |
|---|---|
| read(`n`) | Reads at most `n` characters from the file. Reads till end of file if it is negative or `None`. |
| readable() | Returns `True` if the file stream can be read from. |
| readline(`n`=-1) | Reads and returns one line from the file. Reads in at most `n` bytes if specified. |
| readlines(`n`=-1) | Reads and returns a list of lines from the file. Reads in at most `n` bytes/characters if specified. |
| seek(`offset`,`from`=`SEEK_SET`) | Changes the file position to `offset` bytes, in reference to `from` (start, current, end). |
| seekable() | Returns `True` if the file stream supports random access. |
| tell() | Returns an integer that represents the current position of the file's object. |
| truncate(`size`=`None`) | Resizes the file stream to `size` bytes. If `size` is not specified, resizes to current location. |
| writable() | Returns `True` if the file stream can be written to. |
| write(`s`) | Writes the string `s` to the file and returns the number of characters written. |
| writelines(`lines`) | Writes a list of `lines` to the file. |