

Lecture 1

Introduction to Python

UTAS
Sultanate of Oman
September 2022

CSSY2201 : Introduction to Cryptography

- 1 Python Basics
 - Operators
 - Conditional Statements
 - Loops
 - Built-in Data Types in Python
 - Functions

- 2 Manipulating Files

- 3 Modules

- Manage modules
- Virtual environments

- 4 Data Encoding

- ASCII
- Base64
- utf-8

test the installation of python

Type the following on the shell

```
>>> print('Hello World')
Hello World
>>> x = 100
>>> x*(1 + 0.5)**10
5766.50390625
>>> import math
>>> math.sqrt(49)
7.0
```

Names in Python

Names in Python are **case sensitive** and cannot start with a number. They can contain letters, numbers, and underscores. Some examples include :

- alice
- Alice
- _alice
- _2_alice_
- alice_2

The language includes a number of reserved words, such as `and`, `assert`, `break`, `class`, `continue`, `def`, `del`, `elif`, `else`, `except`, `exec`, `finally`, `for`, `from`, `global`, `if`, `import`, `in`, `is`, `lambda`, `not`, `or`, `pass`, `print`, `raise`, `return`, `try`, `while`



Variables

Values are stored in variables ; the use of a variable is signified by the = sign, which is also known as the assignment operator.

Variables can be overwritten or used in calculations. type into your shell :

```
>>> age + 9
30
>>> age
21
>>> age = age + 9
>>> age
30
```

Strings

Using Strings

```
# use single quotes to define names
>>> name1 = 'John'
>>> name1
'John'
#empty string
>>> strempy = ''
# use double quotes to define names
>>> name2 = "Mary"
>>> name2
'Mary'
```

Strings concatenation with + symbol

```
>>> first = 'John'
>>> last = 'Doe'
>>> name = first + last
>>> name
'JohnDoe'
```

Arithmetic Operators

OPERATOR	DESCRIPTION	EXAMPLE
+	Addition	10 + 20 will give 30
-	Subtraction	10 - 5 will give 5
*	Multiplication	10 * 10 will give 100
/	Division	10 / 2 will give 5
%	Modulus	20 % 10 will give 0
**	Exponent	10**2 will give 100
//	Floor Division	9//2 is equal to 4 and 9.0//2.0 is equal to 4.0

Examples for mod operator % :

EXPRESSION	DESCRIPTION	SYNTAX
28 (mod 26)	28 is equivalent to 2 mod 26	28 % 26
29 (mod 26)	29 is equivalent to 3 mod 26	29 % 26
30 (mod 26)	30 is equivalent to 4 mod 26	30 % 26

Multiplication Operator with the strings

Use the * operator to create specifically formatted strings that may be used in some attacks such as buffer overflows.

```
>>> # Python 3.8
>>> print ('a' * 25)
aaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```


Comparison Operators

also known as relational operators, compare the operands (values) on either side and return true or false based on the condition

OPERATOR	DESCRIPTION	EXAMPLE
==	Compares two operands to see if they are equal; if the values are equal, it returns true.	(10 == 20) is not true
!=	Compares two operands to see if they are not equal; if the values are not equal, it returns true.	(10 != 20) is true
<>	Compares two operands to see if they are not equal; if the values are not equal, it returns true.	(10 <> 20) is true
>	If the operand on the left is greater than the operand on the right, the operation returns true.	(10 > 5) is true
<	If the operand on the right is greater than the operand on the left, the operation returns true.	(10 < 20) is true
>=	If the operand on the right is equal to or less than the value on the left, the condition returns true.	(10 >= 5) is true
<=	If the operand on the left is equal to or less than the value on the right, the condition returns true.	(5 <= 10) is true



Logical Operators

and, or and not

OPERATOR	DESCRIPTION	EXAMPLE
and (logical AND)	If both the operands evaluate to true, then condition becomes true.	(a and b) is true.
or (logical OR)	If any of the two operands are non-zero, then condition becomes true.	(a or b) is true.
not (logical NOT)	Used to reverse the logical state of its operand.	Not(a and b) is false.

Assignment Operators

In addition to using the equal sign, Python offers many assignments that work as a shorthand for more extended tasks.

OPERATOR	DESCRIPTION	EXAMPLE
=	Assigns values from right-side operands to left-side operands.	<code>c = a + b</code> assigns value of <code>a + b</code> into <code>c</code>
<code>+=</code> (add AND)	Adds the right operand to the left operand and assigns the result to the left operand.	<code>c += a</code> is equivalent to <code>c = c + a</code>
<code>-=</code> (subtract AND)	Subtracts the right operand from the left operand and assigns the result to the left operand.	<code>c -= a</code> is equivalent to <code>c = c - a</code>
<code>*=</code> (multiply AND)	Multiplies the right operand with the left operand and assigns the result to the left operand.	<code>c *= a</code> is equivalent to <code>c = c * a</code>
<code>/=</code> (divide AND)	Divides the left operand with the right operand and assigns the result to the left operand.	<code>c /= a</code> is equivalent to <code>c = c / a</code>
<code>%=</code> (modulus AND)	Takes modulus using two operands and assigns the result to the left operand.	<code>c %= a</code> is equivalent to <code>c = c % a</code>
<code>**=</code> (exponent AND)	Performs exponential (power) calculation on operators and assigns the value to the left operand.	<code>c **= a</code> is equivalent to <code>c = c ** a</code>
<code>//=</code> (floor division)	Performs floor division on operators and assigns the value to the left operand.	<code>c //= a</code> is equivalent to <code>c = c // a</code>

Bitwise Operators

The bitwise operators work on bits and perform bit-by-bit operations.

OPERATOR	DESCRIPTION	EXAMPLE
& (binary AND)	Copies a bit to the result if it exists in both operands.	(a & b) (means 0000 1100)
(binary OR)	Copies a bit if it exists in either operand.	(a b) = 61 (means 0011 1101)
^ (binary XOR)	Copies the bit if it is set in one operand but not both.	(a ^ b) = 49 (means 0011 0001)
~ (binary One Complement)	This operator is unary and has the effect of "flipping" bits.	(~a) = -61 (means 1100 0011 in two's complement form due to a signed binary number.
<< (binary Left Shift)	The left operand's value is moved left by the number of bits specified by the right operand.	a << 2 = 240 (means 1111 0000)
>> (binary Right Shift)	The left operand's value is moved right by the number of bits specified by the right operand.	a >> 2 = 15 (means 0000 1111)

Membership Operators

Membership operators test for membership in a sequence, such as strings, lists, or tuples.

OPERATOR	DESCRIPTION	EXAMPLE
<code>in</code>	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	<code>x in y</code> ; here <code>in</code> results in a 1 if <code>x</code> is a member of sequence <code>y</code> .
<code>not in</code>	Evaluates to true if it does not find a variable in the specified sequence and false otherwise.	<code>x not in y</code> ; here <code>not in</code> results in a 1 if <code>x</code> is not a member of sequence <code>y</code> .

Identity Operators

Identity operators compare the memory locations of two objects.

OPERATOR	DESCRIPTION	EXAMPLE
<code>is</code>	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	<code>x is y</code> ; here <code>is</code> results in 1 if <code>id(x)</code> equals <code>id(y)</code> .
<code>is not</code>	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	<code>x is not y</code> ; here <code>is not</code> results in 1 if <code>id(x)</code> is not equal to <code>id(y)</code> .

Python Indentation

- Indentation refers to the spaces at the beginning of a code line.
- Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important.
- Python uses indentation to indicate a block of code.

Example

```
if 5 > 2:  
    print("Five is greater than two!")
```

Python Indentation

- These two examples will generate a syntax error

Syntax Error:

```
if 5 > 2:  
print("Five is greater than two!")
```

Syntax Error:

```
if 5 > 2:  
    print("Five is greater than two!")  
        print("Five is greater than two!")
```


IF, ELSE Statement

conditional statements perform different actions and decide whether a condition is true or false by the `IF` statement

```
If expression
    Statement1
Else
    Statement2
```

To use it in an actual program, type the following

```
>>> for i in range(1,5):
>>>     if i == 2:
>>>         print ('I found two')
>>>     print (i)
```

ELIF

Python also makes use of ELSE and ELIF.

ELSE will capture the execution if the condition is false.

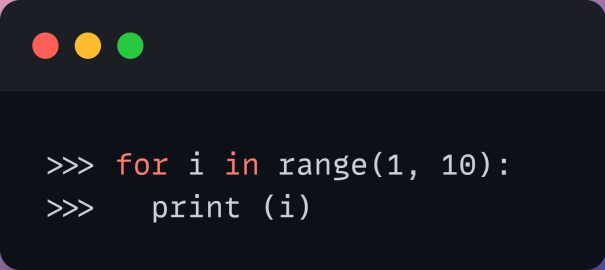
ELIF stands for else if;

```
>>> for i in range(1,5):
>>>     if i == 1:
>>>         print ('I found one')
>>>     elif i == 2:
>>>         print ('I found two')
>>>     elif i == 3:
>>>         print ('I found three')
>>>     else
>>>         print ('I found a number > three')
>>>     print (i)
```

When you use scripting or programming languages, you can perform a set of statements in multiple repetitions. Loops give us the ability to run logic until a specific condition is met

for loop

The for loop is used to iterate over a set of statements that need to be repeated n number of times. The for statement can be used to execute as a counter in a range of numbers. The following syntax prints out the values 1, 2, 3, 4, 5, 6, 7, 8, 9 :



```
>>> for i in range(1, 10):  
>>>     print (i)
```

for loop

The for loop can also execute against the number of elements in a list. Examine the following snippet, which produces the result 76 :

```
>>> numbers = [1, 5, 10, 15, 20, 25]
>>> total = 0
>>> for number in numbers:
>>>     total = total + number
>>> print (total)
```

for loop

the same technique is useful against string arrays. The following outputs three names Eden, Hayden, and Kenna :

```
>>> all_kids = ["Eden", "Hayden", "Kenna"]
>>> for kid in all_kids:
>>>     print(kid)
Eden
Hayden
Kenna
```

while loop

The while loop is used to execute a block of statements while a condition is true. The block of statements may be one or more lines. **The indentation** defines the block. Once a condition becomes false, the execution exits the loop and continues.

```
>>> count = 0
>>> while (count < 5):
>>>     print count
>>>     count = count + 1
>>> print ("The loop has finished.")
```

continue statement

The `continue` statement is used to tell Python to skip the remaining statements in the current loop block and continue to the next iteration. The following snippet will produce an output of 1, 3, 4. The `continue` statement skips printing when `i` equals 2 :

```
>>> for i in range(1,5):  
>>>     if i == 2:  
>>>         continue  
>>>     print (i)
```


break statement

The `break` statement exits out from a loop. The following snippet produces an output of 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11. Once $i = 12$, the loop is abandoned :

```
>>> for i in range(1,15):  
>>>     if i == 12:  
>>>         break  
>>>     print (i)
```

else statement in loops

You can use the `else` statement in conjunction with the `for` and `while` loops to catch conditions that fail. Notice that since the `count==10`, the `while` loop does not execute. You should only see the two final print messages :

```
count=10
while (count < 5):
    print (count)
    count=count+1
else:
    print("count>5")
print("loop finihed")
```

- Binary Types : memoryview, bytearray, bytes
- Boolean Type : bool
- Set Types : frozenset, set
- Mapping Type : dict
- Sequence Types : range, tuple, list
- Numeric Types : complex, float, int
- Text Type : str

You check for the datatype of a variable by the function `type()`

Example	Data type
<code>x = "Hello World"</code>	<code>str</code>
<code>x = 20</code>	<code>int</code>
<code>x=31.6</code>	<code>float</code>
<code>x=2j+3</code>	<code>complex</code>
<code>x = ["apple", "banana", "cherry"]</code>	<code>list</code>
<code>x = ("apple", "banana", "cherry")</code>	<code>tuple</code>
<code>x = range(6)</code>	<code>range</code>
<code>x = {"name" : "John", "age" : 36}</code>	<code>dict</code>
<code>x = {"apple", "banana", "cherry"}</code>	<code>set</code>
<code>x = frozenset({"apple", "banana", "cherry"})</code>	<code>frozenset</code>
<code>x = True</code>	<code>bool</code>
<code>x = b"Hello"</code>	<code>bytes</code>
<code>x = bytearray(5)</code>	<code>bytearray</code>
<code>x = memoryview(bytes(5))</code>	<code>memoryview</code>

Functions are reusable code. You create a new function and assign it a name by using the `def` keyword.

Using an IDLE (ex Spyder) type :

```
def myEnc(plaintext, key):  
    print("ciphertext")
```

Save the file as `MyFunctions.py`. and type in the command line :

```
>>> myEnc('hello', 'secret key')
```

the output should be `ciphertext`

The `def` keyword is used to define the function `myEnc`. It takes 2 arguments and returns an output. We use functions to build logic we intend to use multiple times.

In Python, you can declare some arguments as **optional** :

```
def func(a, b, c=10, d=100):  
    print (a, b, c, d)
```

when tested on the command line :

```
>>> func(1,2)  
1 2 10 100  
>>> func(1,2,3,4)  
1 2 3 4
```

Mode	Explanation
"r"	Read - Default value. Opens a file for reading, error if the file does not exist
"a"	Append - Opens a file for appending, creates the file if it does not exist
"w"	Write - Opens a file for writing, creates the file if it does not exist
"x"	Create - Creates the specified file, returns an error if the file exists

You can specify if the file should be handled as binary or text mode

"t"	Text - Default value. Text mode
"b"	Binary - Binary mode (e.g. images)

Example : `f = open("demofile.txt")`

which is the same as :

`f = open("demofile.txt", "rt")`

Because "r" for read, and "t" for text are the default values, you do not need to specify them.

read files

Use the read() method for reading the content of the file :

```
f = open("demofile.txt", "r")  
print(f.read())
```


You can read the 5 first character of the file :

```
f = open("demofile.txt", "r")  
print(f.read(5))
```

you can return one line by using the readline() method :

```
f = open("demofile.txt", "r") print(f.readline())
```

you can read the whole file, line by line :



```
f = open("demofile.txt", "r")  
for x in f:  
    print(x)
```

Use close() method to close the file after use :

```
f.close()
```


write in a file

To write to an existing file, you must add a parameter to the `open()` function :

- Open the file "demofile2.txt" and append content to the file :

```
f = open("demofile2.txt", "a")  
f.write("Now the file has more content!")  
f.close()
```

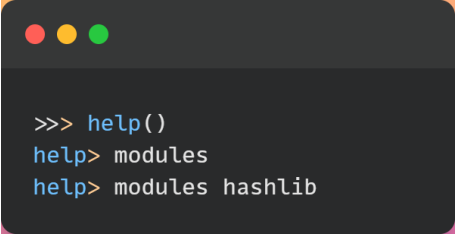
- Open the file "demofile3.txt" and overwrite the content :

```
f = open("demofile3.txt", "w")  
f.write("Woops! I have deleted the content!")  
f.close()
```

- Create a file called "myfile.txt" :

```
f = open("myfile.txt", "x")
```

Python modules are special packages that extend the language. when a module is preinstalled, we can use the `import` command to upload it. To examine the modules that are preinstalled on your system, type the following in the command line :

A terminal window with a dark background and three colored window control buttons (red, yellow, green) at the top left. The text inside the terminal is as follows:

```
>>> help()
help> modules
help> modules hashlib
```

```
help> modules
```

Please wait a moment while I gather a list of all available modules...

```
/nix/store/p21fdyxb3yqflpm7g8s1mymgpnqv7-python3-3.8.12/lib/python3.8/pkgutil.py:92: UserWarning: The numpy.array_api submodule is still experimental. See NEP 47.
```

```
__import__(info.name)
```

```
/home/runner/practical-crypto-course/venv/lib/python3.8/site-packages/_distutils_hack/__init__.py:36: UserWarning: Setuptools is replacing distutils.
```

```
warnings.warn("Setuptools is replacing distutils.")
```

<code>__future__</code>	<code>ast</code>	<code>importlib</code>	<code>sched</code>
<code>_abc</code>	<code>asynchat</code>	<code>importlib_metadata</code>	<code>secrets</code>
<code>_ast</code>	<code>asyncio</code>	<code>inspect</code>	<code>secretstorage</code>
<code>_asyncio</code>	<code>asyncore</code>	<code>io</code>	<code>select</code>
<code>_bisect</code>	<code>atexit</code>	<code>ipaddress</code>	<code>selectors</code>
<code>_blake2</code>	<code>audioop</code>	<code>itertools</code>	<code>setuptools</code>
<code>_bootlocale</code>	<code>backports</code>	<code>jedi</code>	<code>shellingham</code>
<code>_bz2</code>	<code>base64</code>	<code>jeepney</code>	<code>shelve</code>
<code>_cffi_backend</code>	<code>bcrypt</code>	<code>json</code>	<code>shlex</code>
<code>_codecs</code>	<code>bdb</code>	<code>keyring</code>	<code>shutil</code>
<code>_codecs_cn</code>	<code>binascii</code>	<code>keyword</code>	<code>signal</code>
<code>_codecs_hk</code>	<code>binhex</code>	<code>lect2</code>	<code>site</code>
<code>_codecs_iso2022</code>	<code>bisect</code>	<code>lib2to3</code>	<code>six</code>
<code>_codecs_jp</code>	<code>builtins</code>	<code>libfuturize</code>	<code>smtpd</code>
<code>_codecs_kr</code>	<code>bz2</code>	<code>libpasteurize</code>	<code>smtplib</code>
<code>_codecs_tw</code>	<code>cProfile</code>	<code>linecache</code>	<code>sndhdr</code>
<code>_collections</code>	<code>cachecontrol</code>	<code>locale</code>	<code>socket</code>
<code>_collections_abc</code>	<code>cachy</code>	<code>lockfile</code>	<code>socketserver</code>
<code>_compat_pickle</code>	<code>calendar</code>	<code>logging</code>	<code>spwd</code>
<code>_compression</code>	<code>certifi</code>	<code>lzma</code>	<code>sqlite3</code>
<code>_contextvars</code>	<code>cffi</code>	<code>mailbox</code>	<code>sre_compile</code>



Example of a module

The hashlib is a built-in module that is preinstalled.

Type `import hashlib`, then in the terminal type `hashlib`. (enter the dot after `hashlib`). You should see a list of methods.

Let's try one :

```
>>> import hashlib
>>> hashlib.md5('hello world'.encode()).hexdigest()
'5eb63bbbe01eeed093cb22bb8f5acdc3'
>>> hashlib.sha512('hello world'.encode()).hexdigest()
'309ecc489c12d6eb4cc40f50c902f2b4d0ed77ee511a7c7a9bcd3ca86d4cd86f
989dd35bc5ff499670da34255b45b0cfd830e81f605dcf7dc5542e93ae9cd76f'
```

install a module

- Use `pip` command on terminal : The `pip` command looks for the package in PyPI, resolves its dependencies, and installs everything in your current Python environment to ensure that requests will work. The `pip install <package>` command always looks for the latest version of the package and installs it

or

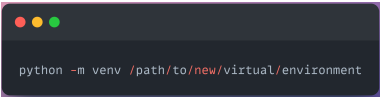
- Use `conda` command on terminal OR Anaconda Navigator if Anaconda is installed : The `conda` command is the primary interface for managing installations of various packages. It can Query and search the Anaconda package index and current Anaconda installation. Create new conda environments. Install and update packages into existing conda environments. The navigator is just an extension for conda command.

Virtual environments

- A virtual environment is a Python environment such that the Python interpreter, libraries and scripts installed into it are isolated from those installed in other virtual environments, and (by default) any libraries installed in a "system" Python, i.e., one which is installed as part of your operating system.
- A virtual environment is a directory tree which contains Python executable files and other files which indicate that it is a virtual environment.
- Common installation tools such as `setuptools` and `pip` work as expected with virtual environments. In other words, when a virtual environment is active, they install Python packages into the virtual environment without needing to be told to do so explicitly.

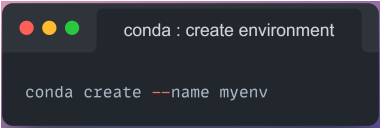
Create Virtual environments

- The venv module provides support for creating isolated "virtual environments" with their own site directorie. Each environment has its own Python binary and can have its own independent set of installed Python packages in its site directories.



```
python -m venv /path/to/new/virtual/environment
```

- if you are using Anaconda. Use the Anaconda Navigator to create and manage environments or the conda command :



```
conda create --name myenv
```

ASCII encoding

- ASCII, abbreviated from American Standard Code for Information Interchange, is a character-encoding standard for electronic communication. ASCII codes represent text in computers, telecommunications equipment, and other devices. Most modern character-encoding schemes are based on ASCII, although they support many additional characters.
- In ASCII encoding, each letter is converted to one byte. Look at the following examples :

$A = 65 \text{ or } 0b01000001$

$B = 66 \text{ or } 0b01000010$

$C = 67 \text{ or } 0b01000011$

$ABC = 0b01000001 \ 0b01000010 \ 0b01000011$

Base64 Encoding Text

- Base64, also known as privacy enhanced electronic mail (PEM), is the encoding that converts binary data into a textual format ; it can be passed through communication channels where text can be handled in a safe environment. PEM is primarily used in the email encryption process. To use the functions included in the Base64 module, you will need to import the library in your code. Base64 offers a decode and encode module that both accepts input and provides output.
- To break ASCII encoding into Base64-encoded text, each sequence of six bits encodes to a single character.

Base64 table

Index	Binary	Char	Index	Binary	Char	Index	Binary	Char	Index	Binary	Char
0	000000	A	16	010000	Q	32	100000	g	48	110000	w
1	000001	B	17	010001	R	33	100001	h	49	110001	x
2	000010	C	18	010010	S	34	100010	i	50	110010	y
3	000011	D	19	010011	T	35	100011	j	51	110011	z
4	000100	E	20	010100	U	36	100100	k	52	110100	0
5	000101	F	21	010101	V	37	100101	l	53	110101	1
6	000110	G	22	010110	W	38	100110	m	54	110110	2
7	000111	H	23	010111	X	39	100111	n	55	110111	3
8	001000	I	24	011000	Y	40	101000	o	56	111000	4
9	001001	J	25	011001	Z	41	101001	p	57	111001	5
10	001010	K	26	011010	a	42	101010	q	58	111010	6
11	001011	L	27	011011	b	43	101011	r	59	111011	7
12	001100	M	28	011100	c	44	101100	s	60	111100	8
13	001101	N	29	011101	d	45	101101	t	61	111101	9
14	001110	O	30	011110	e	46	101110	u	62	111110	+
15	001111	P	31	011111	f	47	101111	v	63	111111	/
Padding		=									

- Examine the 24 bits from the previous section :

0b01000001 0b01000010 0b01000011

- Break the line into 6-bit groups :

0b010000 010100 001001 000011

- When you convert the four groups to decimal, you will see that they are equal to the following :

16 20 9 3

- You now convert the numbers to Base64 :

QUJD

- Therefore, when you encode "ABC" to Base64, you should end up with QUJD, as shown here :

```
>>> import base64
>>> value = 'ABC'.encode()
>>> print(base64.b64encode(value))
b'QUJD'
```

- In the event that the text cannot be broken down into groups of six, you will see the padding character, which is shown using the equal sign =. If the example had four bytes : for example 'ABCD', then the output : 'QUJDRA==' :

```
In [4]: import base64
In [5]: value='ABCD'.encode()
In [6]: print(base64.b64encode(value))
b'QUJDRA=='
```

utf-8

- UTF-8 is a variable-width character encoding used for electronic communication. Defined by the Unicode Standard, the name is derived from Unicode (or Universal Coded Character Set) Transformation Format 8-bit.
- UTF-8 is capable of encoding all 1,112,064 valid character code points in Unicode using one to four one-byte (8-bit) code units. Code points with lower numerical values, which tend to occur more frequently, are encoded using fewer bytes. It was designed for backward compatibility with ASCII : the first 128 characters of Unicode, which correspond one-to-one with ASCII, are encoded using a single byte with the same binary value as ASCII, so that valid ASCII text is valid UTF-8 encoded Unicode as well.

utf-8

- When dealing with displaying ciphertexts and hashes (digests), the output is binary (raw bytes). when you try to display this what you get :

```
>>>import hashlib
>>>plaintext_password = b'password'
>>>hashed = hashlib.md5(plaintext_password).digest()
>>>print(hashed)

b"_M\xcc;Z\xa7e\xd6\x1d\x83'\xde\xb8\x82\xcf\x99"
```

- to be readable, convert it to hexadecimal or base64 like follows :

```
>>>import hashlib
>>>plaintext_password = b'password'
>>>hashedHEX = hashlib.md5(plaintext_password).hexdigest()
>>>print(hashedHEX)

5f4dcc3b5aa765d61d8327deb882cf99
```

utf-8

- or simply produce the binary digest and use *hex* and *base64.b64encode* functions to produce the hexadecimal and base64 representation of the digest :

```
import hashlib
import base64
plaintext_password = b'password'
hashed = hashlib.md5(plaintext_password).digest()
print("binary digest =", hashed)
hhex=hashed.hex()
print("hexadecimal digest =", hhex)
encoded = base64.b64encode(hashed)
print("base64 digest =", encoded)
```

- result will be the same hash displayed in binary, hexadecimal, and base64 encoding.

```
~/practical-crypto-course$ python lect2.py
binary digest = b'_M\xcc;Z\xa7e\xd6\x1d\x83'\xde\xb8\x82\xcf\x99"
hexadecimal digest = 5f4dcc3b5aa765d61d8327deb882cf99
base64 digest = 'X03M01qnZdYdgyfeuILPmQ=='
```