محاضــرات في

مادة البرمجة والنمذجة

(314 ا)

للفرقــة الثالثــة

مساحة ونظم معلومات

جغــرافيـــة

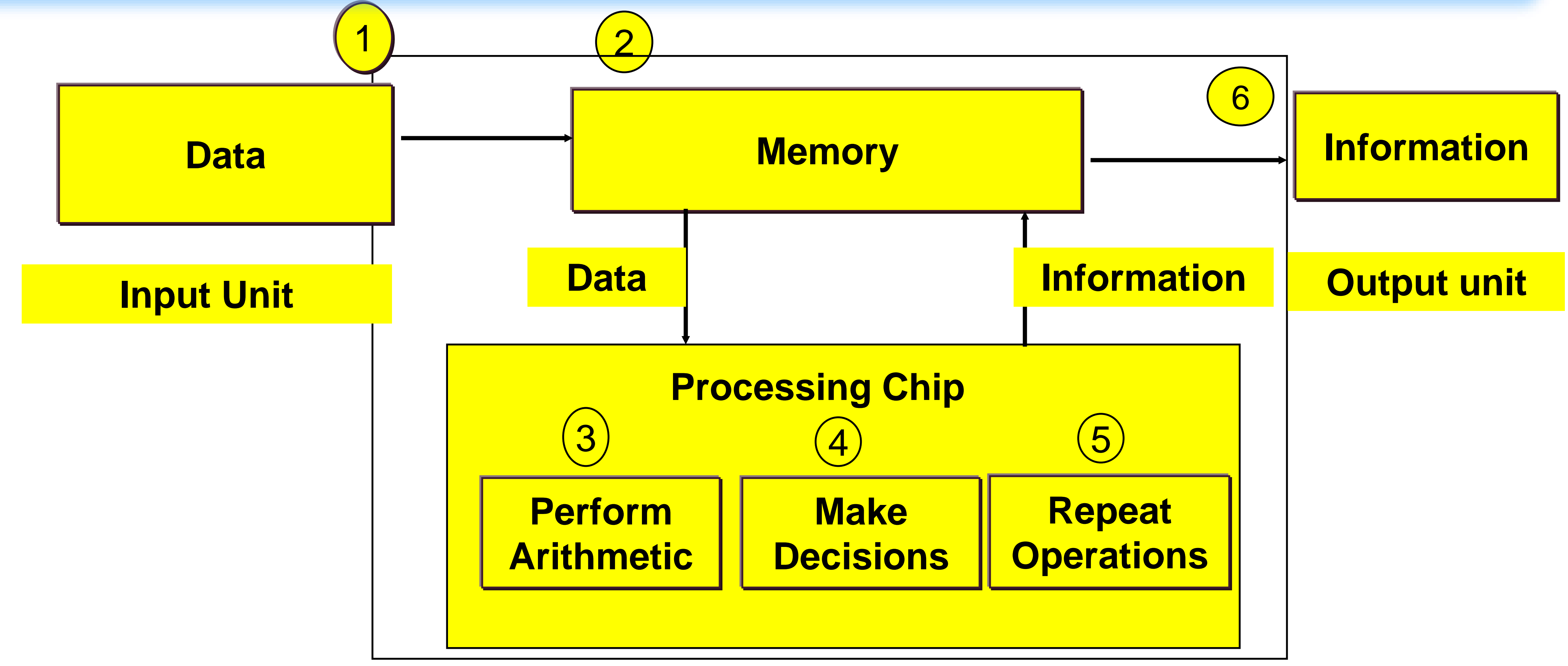
Programming Fundamentals Python

Python Programming Fundamentals

Python Programming Fundamentals

Data Types & Variables

Operations Performed by the Computer



البـــرمجـــــــة

هى طريقة لحل المسائل تهدف الى تقديم الحل فى صورة خطوات مرتبة ترتيبا منطقيا اذا تتبعناه نصل الى حل المسألة

هذه الخطوات يطلق عليها Algorithm

حـــل المشكلات بالكمبيوتر

خطوات حل المشكلة يطلق عليها **Algorithm** ويجب ان يتم تحويل هذه الخطوات الى برنامج **Program** باستخدام احد لغات البرمجة

Problem ⟶ **Algorithm** ⟶ **Program**

Algorithm [Rule, Procedure, Method, Technique]

Designing a program عمليات حل المشكلة

1. Analyzing The Problem

2. Developing The Algorithm

Design a solution / program

3. Coding The Program (Basic)

(Suitable Programming Lang).

عمليـــات حـــل المشكلـــة

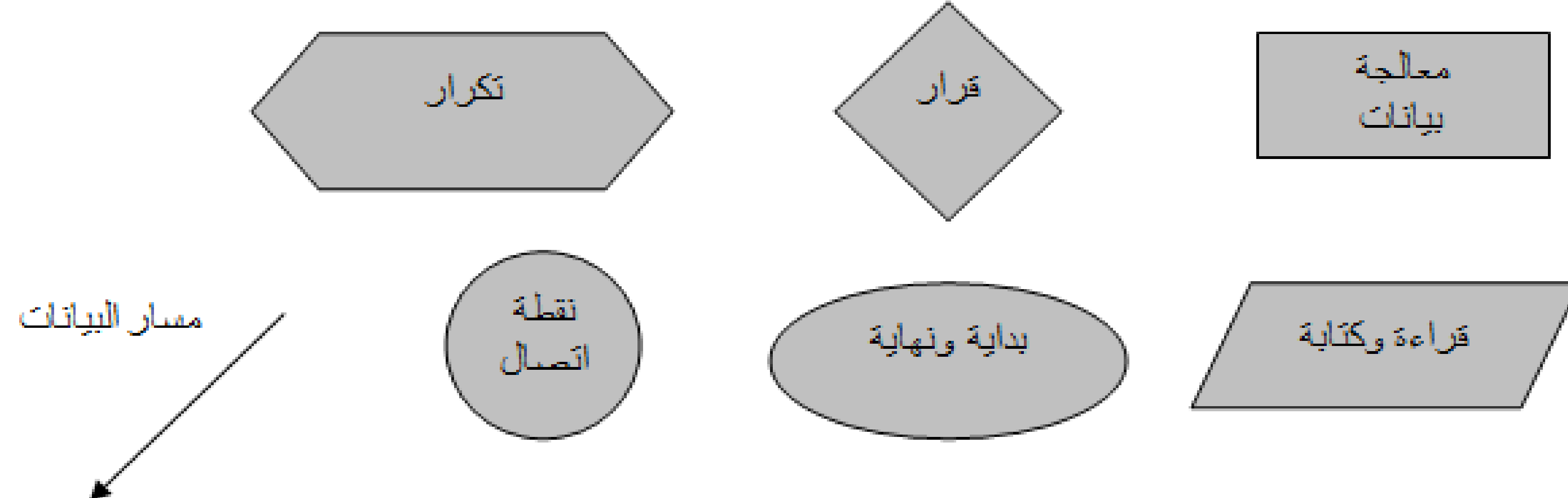
4. Executing The Program

5. Testing The Program

6. Documenting The Program

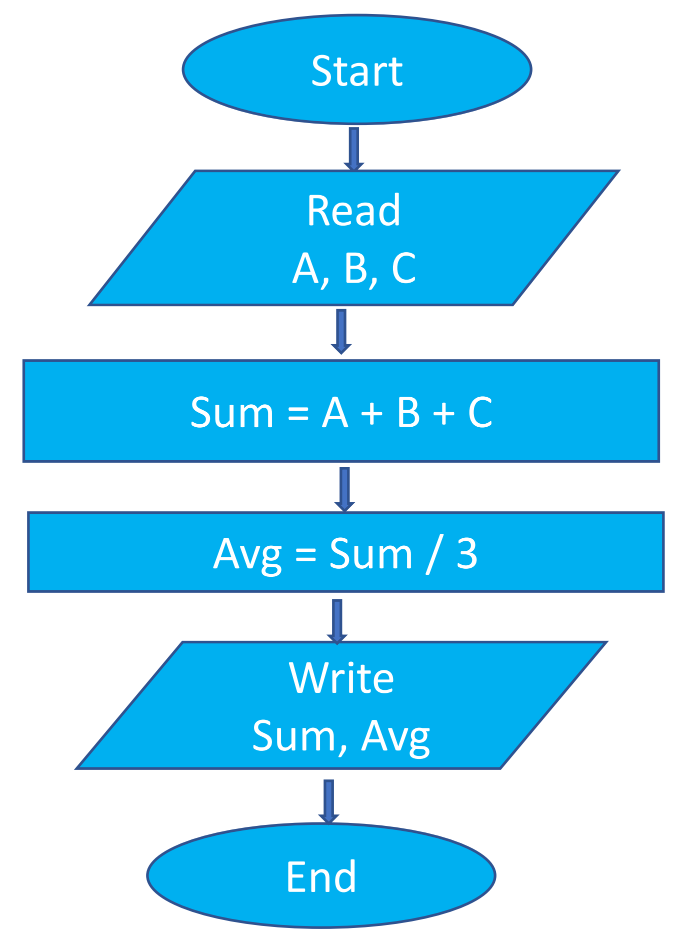
FLOWCHART

Flowchart is a diagrammatic way of representing, the steps to be followed for solving the given problem, and provides us the visualization of the steps involved



أمثلة لخرائط التدفق Flowchart

مثال ارسم خريطة التدفق لحساب وطباعة المجموع والمتوسط ل 3 اعداد A,B,C . ثم حولها الى Pseudocode



BEGIN

DISPLAY „input 3 nos“

INPUT A, B, C

DETERMINE SUM = A + B + C

DETERMINE AVG = SUM / 3

PRINT SUM, AVG

END

- **Created in 1991 by Guido van Rossum (now at Google)**
  - **Named for Monty Python**

- Used by:
  - Google, Yahoo!, Youtube
  - Many Linux distributions
  - Games and apps (e.g. Eve Online)

Interpreted Languages

- **Interpreted**
  - Not compiled like Java
  - Code is written and then directly executed by an **interpreter**
  - Type commands into interpreter and see immediate results

Python:

Code	Interpreter	Computer

The Python Interpreter

- Allows you to type commands one-at-a-time and see results
- A great way to explore Python's syntax
  - Repeat previous command: Alt+P

Python Shell

File Edit Shell Debug Options Windows Help

```
Python 2.4.3 (#69, Mar 29 2006, 17:35:34) [MSC v.1310 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.

    ****************************************************************
    Personal firewall software may warn about the connection IDLE
    makes to its subprocess using this computer's internal loopback
    interface.  This connection is not visible on any external
    interface and no data is sent to or received from the Internet.
    ****************************************************************

IDLE 1.1.3
>>> print "Hello there"
Hello there
>>> print "How are you"
How are you
>>> |
```

Ln: 16 Col: 4

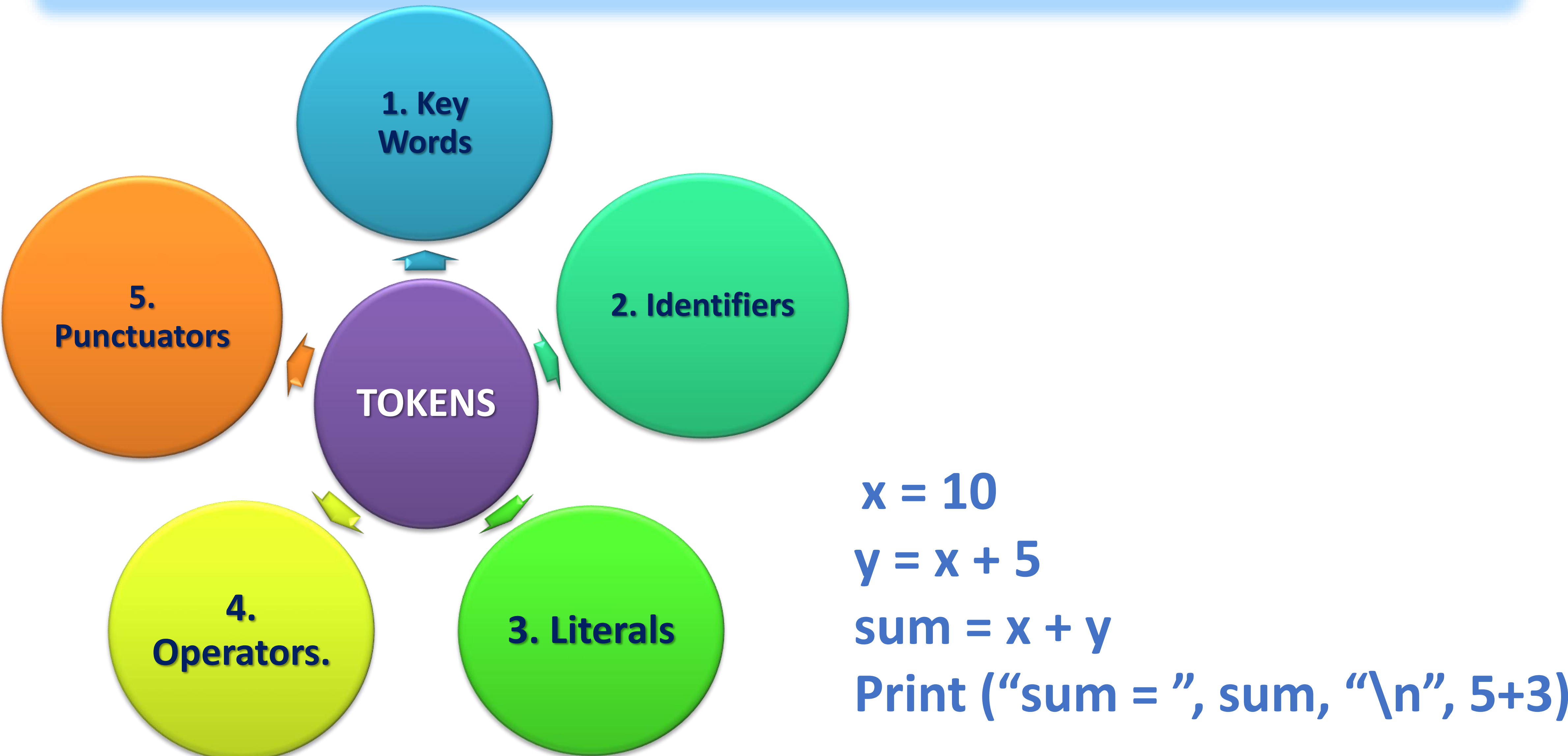
Our First Python Program

- Python does not have a `main` method like Java
  - The program's main code is just written directly in the file
- Python statements do not end with semicolons (;)

hello.py

```
1  print("Hello, world!")
```

TOKENS / LEXICAL UNITS



Identifiers

- **Python has some rules about how identifiers can be formed**
  - **Every identifier must begin with a letter or underscore, which may be followed by any sequence of letters, digits, or underscores**

```
>>> x1 = 10
>>> x2 = 20
>>> y_effect = 1.5
>>> celsius = 32
>>> 2celsius
  File "<stdin>", line 1
    2celsius
           ^
SyntaxError: invalid syntax
```

Identifiers

- **Python has some rules about how identifiers can be formed**
  - **Identifiers are *case-sensitive***

```
>>> x = 10
>>> X = 5.7
>>> print(x)
10
>>> print(X)
5.7
```

Identifiers (keywords)

- **Python has some rules about how identifiers can be formed**
  - **Some identifiers are part of Python itself (they are called *reserved words* or *keywords*) and cannot be used by programmers as ordinary identifiers**

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

Python Keywords

Identifiers

- **Python has some rules about how identifiers can be formed**
  - **Some identifiers are part of Python itself (they are called *reserved words* or *keywords*) and cannot be used by programmers as ordinary identifiers**

An example…

```
>>> for = 4
  File "<stdin>", line 1
    for = 4
        ^
SyntaxError: invalid syntax
```

Literals

- In the following example, the parameter values passed to the print function are all technically called *literals*

```
>>> print("Hello")
Hello
>>> print("Programming is fun!")
Programming is fun!
>>> print(3)
3
>>> print(2.3)
2.3
```

- ❑ More precisely, “Hello” and “Programming is fun!” are called *textual literals*, while 3 and 2.3 are called *numeric literals*

PUNCTUATORS

- **Punctuators are also called as separators**
- **The Followings are used as punctuators:**

- Brackets [ ]
- Parentheses ( )
- Braces { }
- Comma ,
- Semicolon ;

- Colon :
- Asterisk *
- Ellipsis ...
- Equal Sign =
- Pound Sign #

Python Comments

- **A python comment begins with a "#".**
- **Anything after the "#" is ignored by Python**
- **The 1st line in the below script is a comment line- it is ignored by Python**
- **The Characters to the right of the "#" on lines 2-5 are ignored**

```
# get x1, y1, x2, y2 from the command line
x1param = float(5)   # x1
y1param = int(5.23)   # y1

X2param?= float(y1   n)   # x2
y2pa ram = flo   param)   # y2
2z3param =    (y1param)    # z3
```

Invalid Variabel

Python Data Types

- **String:** a sequence of alphanumeric characters
- **Integer**: a whole number that has no fractional component
- **Float:** a number that contains a fractional component
- **String** example: “I learn Python” (note that strings are enclosed in quotes)
- **Integer** examples: 100, -19, 0, 9999999
- **Float** examples: 1.0, -123.678, 1.6745E3

Python Assignment Statement

- The “=“ sign is the assignment operator as it is in most programming languages

```
X = 1
print  (X)     # the number “1” will appear on the screen
X = X + 5
print (X)     # the number “6” will appear on the screen
```

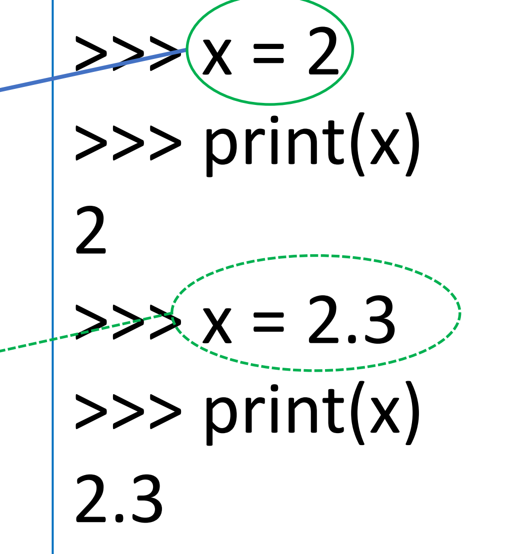
Interactive Window

```
>>> x = 1
>>> print x
1
>>> x = x + 5
>>> print x
6
>>>
```

Simple Assignment Statements

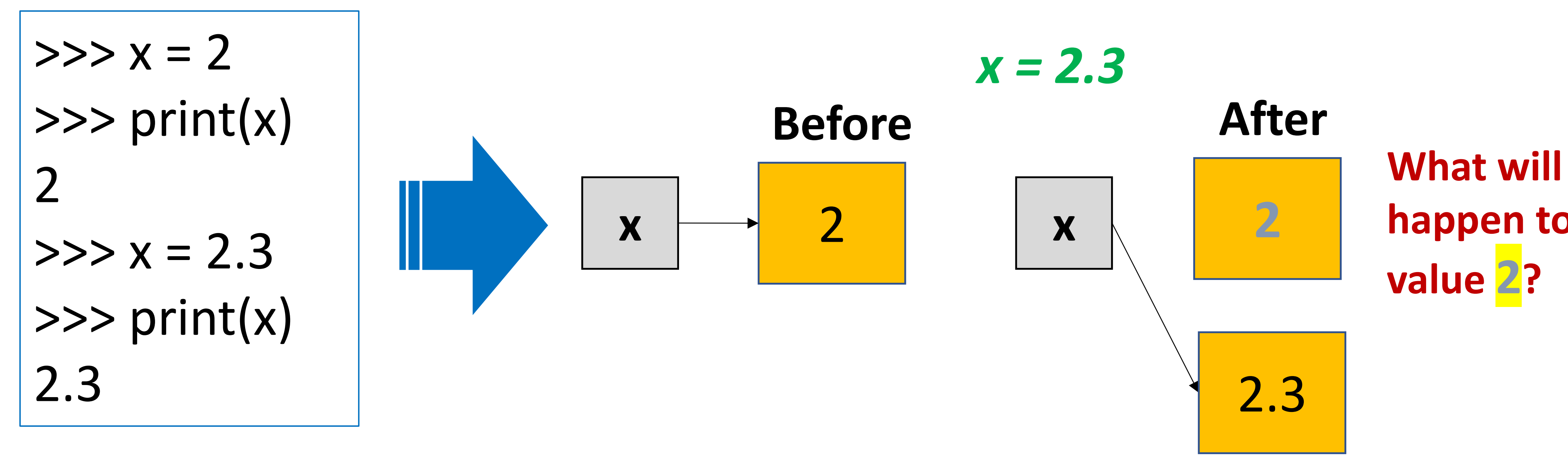
- A literal is used to indicate a specific value, which can be *assigned* to a *variable*

  - **x is a variable and 2 is its value**
  - **x can be assigned different values; hence, it is called a variable**



Simple Assignment Statements: Actual View

- **Python assignment statements are actually slightly different from the “Variable as a Box” model.**
  - **In Python, values may end up anywhere in memory, and variables are used to refer to them.**



Assignning Input

- So far, we have been using values specified by programmers and printed or assigned to variables
  - How can we let users (not programmers) input values?

- In Python, input is accomplished via an assignment statement combined with a built-in function called *input*

```
<variable> = input(<prompt>)
```

- When Python encounters a call to *input*, it prints <prompt> (which is a string literal) then pauses and waits for the user to type some text and press the <Enter> key

Assignning Input

- Here is a sample interaction with the Python interpreter:

```
>>> name = input("Enter your name: ")
Enter your name:    Abdou Hussien
```

```
>>> name
‘Abdou Hussien'
>>>
```

- Notice that whatever the user types is then stored as a String
  - What happens if the user inputs a number?

Assignning Input

- Here is a sample interaction with the Python interpreter:

```
>>> number = input("Enter a number: ")
Enter a number: 3
>>> number
'3'
>>>
```

Still a String!

- How can we force an input number to be stored as a number and not as a string?
  - We can use the built-in ***eval*** function, which can be “Wrapped Around” the input function.

Assignning Input

- **Here is a sample interaction with the Python interpreter:**

```
>>> number = eval (input ("Enter a number: "))
Enter a number: 3
>>> number
3
>>>
```

Now an int (no single quotes)!

Assignning Input

- **Here is a sample interaction with the Python interpreter:**

```
>>> number = eval (input("Enter a number: "))
Enter a number: 3.7
>>> number
3.7
>>>
```

And now a float (no single quotes)!

Assignning Input

- **Here is another sample interaction with the Python interpreter:**

```
>>> number = eval (input("Enter an equation: "))
Enter an equation: 3 + 2
>>> number
5
>>>
```

The *eval* function will evaluate this formula and return a value, which is then assigned to the variable “number”

Data Type Conversion

- **Besides, we can convert the string output of the *input* function into an integer or a float using the built-in *int* and *float* functions**

```
>>> number = int (input("Enter a number: "))
Enter a number: 3
>>> number
3
>>>
```

An integer (no single quotes)!

Data Type Conversion

- **Besides, we can convert the string output of the *input* function into an integer or a float using the built-in *int* and *float* functions**

```
>>> number = float(input("Enter a number: "))
Enter a number: 3.7
>>> number
3.7
>>>
```

A float (no single quotes)!

Data Type Conversion

- **As a matter of fact, we can do various kinds of conversions between strings, integers and floats using the built-in *int*, *float*, and *str* functions:**

```
>>> x = 10
>>> float(x)
10.0
>>> str(x)
'10'
>>>
```

integer → float
integer → string

```
>>> y = "20"
>>> float(y)
20.0
>>> int(y)
20
>>>
```

string → float
string → integer

```
>>> z = 30.0
>>> int(z)
30
>>> str(z)
'30.0'
>>>
```

float → integer
float → string

Simultaneous Assignment

- Python allows us also to assign multiple values to multiple variables all at the same time

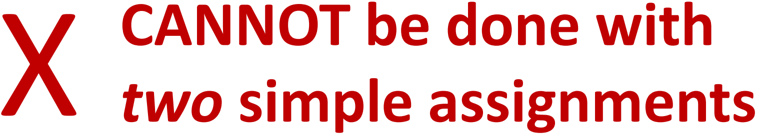
```
>>> x, y = 2, 3
>>> x
2
>>> y
3
>>>
```

- This form of assignment might seem strange at first, but it can prove remarkably useful (e.g., for swapping values)

Simultaneous Assignment

- **Suppose you have two variables x and y, and you want to swap their values (*i.e., you want the value stored in x to be in y and vice versa*)**

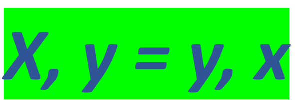
```
>>> x = 2
>>> y = 3
>>> x = y
>>> y = x
>>> x
3
>>> y
3
```

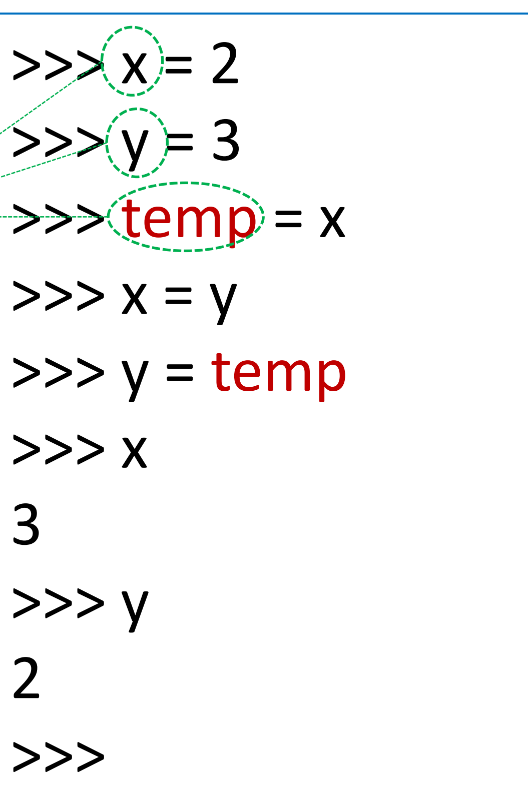


Simultaneous Assignment

- **Suppose you have two variables x and y, and you want to swap their values (*i.e., you want the value stored in x to be in y and vice versa*)**

Thus far, we have been using different *names* for variables. These names are technically called ***identifiers***





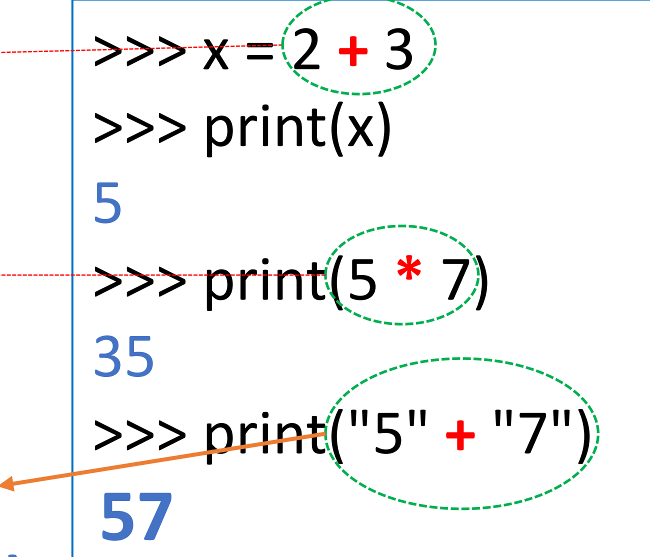
CAN be done with *three* simple assignments, but more efficiently with simultaneous assignment

Expressions

- **You can produce new data (numeric or text) values in your program using *expressions***

- **This is an expression that uses the *addition operator***
- **This is another expression that uses the *multiplication operator***

- **This is another expression that uses the addition operator but to concatenate (or glue) strings together**



Expressions

- **You can produce new data (numeric or text) values in your program using *expressions***

Another example…

```
>>> x = 6
>>> y = 2
>>> print(x - y)
4
>>> print(x/y)
3.0
>>> print(x//y)
3
```

Yet another example…

```
>>> print(x*y)
12
>>> print(x**y)
36
>>> print(x%y)
0
>>> print(abs(-x))
6
```

Expressions: Summary of Operators

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Float Division
**	Exponentiation
abs()	Absolute Value
//	Integer Division
%	Remainder

Python Operators (in order of precedence)

1. Brackets: ( )
2. Multiplication: *
3. Division: /
4. Modulus: %
5. Addition: +
6. Subtraction: -

Explicit and Implicit Data Type Conversion

- Data conversion can happen in two ways in Python
  1. **Explicit Data Conversion** (we saw this earlier with the *int*, *float*, and *str* built-in functions)
  2. **Implicit Data Conversion**
     - Takes place *automatically* during run time between *ONLY* numeric values
       - E.g., Adding a float and an integer will automatically result in a float value
       - E.g., Adding a string and an integer (or a float) will result in an *error* since string is not numeric
     - Applies *type promotion* to avoid loss of information
       - Conversion goes from integer to float (e.g., upon adding a float and an integer) and not vice versa so as the fractional part of the float is not lost

Implicit Data Type Conversion: Examples

- **The result of an expression that involves a float number alongside (an) integer number(s) is a float number**

```
>>> print(2 + 3.4)
5.4
>>> print( 2 + 3)
5
>>> print(9/5 * 27 + 32)
80.6
>>> print(9//5 * 27 +
32)
59
>>> print(5.9 + 4.2)
10.100000000000001
>>>
```

Implicit Data Type Conversion: Examples

- **The result of an expression that involves a float number alongside (an) integer number(s) is a float number**
- **The result of an expression that involves values of the same data type will not result in any conversion**

```
>>> print(2 + 3.4)
5.4
>>> print( 2 + 3)
5
>>> print(9/5 * 27 + 32)
80.6
>>> print(9//5 * 27 + 32)
59
>>> print(5.9 + 4.2)
10.100000000000001
>>>
```

Built-in Python Functions

- A function takes an "argument" or "arguments" and returns a value that can be used in an assignment statement
- In the below satements abs(x) and pow(x,y) are built-in functions in every implementation of the Python language

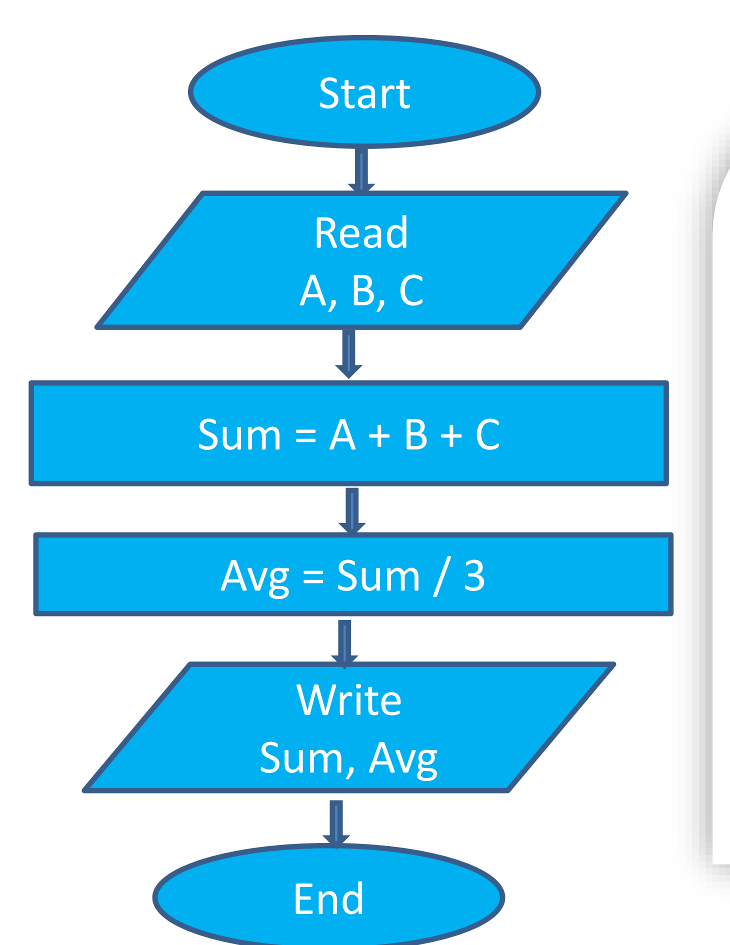
```
x = abs(-8)
print (x )          # the number "8" will appear

y = pow(3,2)
print (y )          # the number "9" will appear
```

Python Built-In Functions

- **abs(x)** # returns the absolute value of x
- **float(x)** # returns the string x converted to a floating point number
- **int(x)** # returns the string x converted to a integer number
- **pow(x,y)** # returns the number x rasied to the y power
- **round(x,n)** # rounds the number x to n decimal places
- **str(x)** # returns the string equivalent of the object x

Programm

مثال اكتب برنامجا لحساب وطباعة المجموع والمتوسط ل 3 اعداد A,B,C .




```
1 # write Python Programe to caculate and print the mean of 3 Numbers, A, B, C.
```

```
1 # this Programe calculate the mean of 3 Numbers
2
3 A = 5
4 B = 8
5 C = 8
6 summ = A + B + C
7 Avrg = summ/3
8
9 print ('Summation = ', summ, '  The Average is : ', Avrg)
```

```
Summation =  21   The Average is :  7.0
```

Summary

- Programs are composed of statements that are built from *identifiers* and *expressions*

- Identifiers are names
  - They begin with an underscore or letter which can be followed by a combination of letter, digit, and/or underscore characters
  - They are case sensitive. (e.g.: a , a42, a4xy, name, user_name, username, ….)

- Expressions are the fragments of a program that produce data
  - They can be composed of *literals*, *variables*, and *operators*

Summary

- A literal is a representation of a specific value (e.g., **3** is a literal representing the number **three**)

- A variable is an identifier that stores a value, which can change (hence, the name *variable*)

- Operators are used to form and combine expressions into more complex expressions (e.g., the expression **x + 3 * y** combines two expressions together using the **+** and ***** operators)

Summary

- In Python, *assignment* of a value to a variable is done using the equal sign (i.e., =)
- Using assignments, programs can get inputs from users and manipulate them internally
- Python allows *simultaneous assignments*, which are useful for swapping values of variables
- *Datatype conversion* involves converting *implicitly* and *explicitly* between various datatypes, including integer, float, and string

Programming Fundamentals Python

Coding with Python

Simple Data Types

- **Numbers**
  - Integer, Floating-point, Complex!   # c=complex(12,5)
    # c = 12 + 5j
- **Strings**
  - Characters are strings of length 1
- **Booleans** are `False` or `True`

Simple Data Types: Operators

- `+, -, *, /, %` # x + y, x-y, x*y, x/y
- `+=, -=, …` etc. # x += 2 ==➔ x = x + 2
- Assignment using `=`
  but semantics are different!
  ```
  a = 1        # int
  a = "food"   # str
  ```
- Can also use `+` to concatenate strings
  - "hello " + "world" ➔ "hello world"

A quick note on the increment operator shorthand

- Python has a common idiom that is not necessary, but which is used frequently and is therefore worth noting:

  `x += 1` Is the same as `x = x + 1`

- This also works for other operators:

```
x += y        # adds y to the value of x
x *= y        # multiplies x by the value y
x -= y        # subtracts y from x
x /= y        # divides x by y
```

Boolean Operators

- Boolean operators are useful when making conditional statements, we will cover these in-depth later.
- and
- or
- not

Comparison Operators

- Greater than: **>**
- Lesser than: **<**
- Greater than or equal to: **>=**
- Lesser than or equal to: **<=**
- Is equal to: **==**

- **Write a couple of operations using comparison operators; i.e.**

In:

```
intVar = 5
floatVar = 3.2
stringVar = "Food"

if intVar > floatVar:
    print("Yes")

if intVar == 5:
    print("A match!")
```

Out:

```
Yes
A match!
```

String

- "hello" + "world" — "helloworld" # concatenation
- "hello" * 3 — "hellohellohello" # repetition
- "hello" [0] — "h" # indexing
- "hello" [-1] — "o" # (from end)
- "hello" [1:4] — "ell" # slicing
- **Len** ("hello") — 5 # size
- "hello" < "jello" — True # comparison
- "e" **in** "hello" — True # search
- New line: — "escapes: \n "
- Line continuation: — triple quotes """ """
- Quotes: — 'single quotes', " double quotes "

Methods in string

- upper()
- lower()
- count(s) # text.count('word')
- find(s)
- index(s) # s.index('l')

Simple Data Types

- Triple quotes useful for multi-line strings

```
s = """a long string with "quotes" or anything else"""

  print(s)
'a long string with "quotes" or anything else '

  >>> len(s)
  44
```

Compound Data Type: List

- List:
  - Collection allows us to put many values in a single "variable"
  - Defined in square brackets

```
a = [1, 2, 3, 4, 5]
print a[1]   # number 2
some_list = [     ]
some_list.append("food")
some_list.append(12)# some_list = ["food", 12 ]
print len(some_list)         # 2
friends = ['Ahmed', 'Ali', 'Yasser','Sally']
```

Compound Data Type: List

- *a = [99, "bottles of beer", ["on", "the", "wall"]]*

  a [2] =======➔ *["on", "the", "wall"]*

Flexible arrays

- Same Operators as for strings
  - a + b, a*3, a[0], a[-1], a[1:], len(a)
- Item and slice assignment
  - *a[0] = 98*
  - *a[1:2] = ["bottles", "of", "beer"]*

    a ➔ *[98, ["bottles", "of", "beer“], ["on", "the", "wall"]]*
  - *del a[-1] # -> [98, ["bottles", "of", "beer“]]*

Compound Data Type: List

```
>>> a = [x for x in range(5)]     # a = [0,1,2,3,4]
>>> a.append(5)                   # [0,1,2,3,4,5]
>>> a.pop()                       # [0,1,2,3,4]
5
>>> a.insert(0, 5.5)              # [5.5,0,1,2,3,4]
>>> a.pop(0)                      # [0,1,2,3,4]
5.5
>>> a.reverse()                   # [4,3,2,1,0]
>>> a.sort()                      # [0,1,2,3,4]
```

Nested List

- List in a list
- E.g.,
  - >>> s = [1,2,3]
  - ~~>>> t = [‘begin’, S, ‘end’]~~ Error
  - >>> t = [‘begin’, s, ‘end’]
  - >>> t
  - [‘begin’, [1, 2, 3], ‘end’]
  - >>> t[1][1]
  - 2

An example

- **We have a list of species:**

```
species = ['dog', 'cat', 'shark', 'falcon', 'deer', 'tyrannosaurus rex']
for i in species:
    print(i)
```

- The command underneath the list then cycles through each entry in the species list and prints the animal's name to the screen. Note: The i is quite arbitrary. You could just as easily replace it with 'animal', 't', or anything else.

```
In [1]: runfile('//is
dog
cat
shark
falcon
deer
tyrannosaurus rex
```

For loops essentially say:

"For all elements in a sequence, do something"

Another example

- We can also use for loops for operations other than printing to a screen. For example:

```
numbers = [1, 20, 18, 5, 15, 160]
total = 0
for value in numbers:
    total = total + value
print(total)
```

```
In [4]: runfile
219
```

- Using the list you made a moment ago, use a for loop to print each element of the list to the screen in turn.

Applications [Variables]

<var_name> = <value> # ➔ age , name = 22, "Ahmed"

grades = [67, 100, 87, 56]

complex(4, 5) # (4+5j)

print("Hello, World!")

complex(0, 0) # 0j

"I'm 20 years old" or 'My favorite book is "Sense and Sensibility"'

type (varaibleName)

String: H e l l o

Index: 0 1 2 3 4

<string_variable>[start:stop:step]

Applications [Variables]

```
>>> freecode = "freeCodeCamp"
```

```
freecode . replace ("e", "a")  # 'fraaCodaCamp'
```

```
>>> freecode.capitalize()     # 'Freecodecamp'

>>> freecode.count("C")      #  2

>>> freecode.index("p")      # 11

>>> freecode.isalpha()       # True

>>> freecode . islower()     #  False

>>>freecode . isspace ()       #  False
```

```
>>> freecode.split("C")
['free', 'ode', 'amp']

>>> freecode.swapcase()
'FREEcODEcAMP'

>>> freecode.title()
'Freecodecamp'

>>> freecode.upper()
'FREECODECAMP'
```

Nested Lists

List = [1, 2, 3, 4, 5, "a", "b", "c", 3.4, 2.4, 2.6]

Nested_list = [[1, 2, 3], [4, 5, 6]] # Nested List

List Length # len(List)

Update a Value in a List # list[0] = 'H'

Add a Value to a List # list.append(10)

list.insert (5, 6)

List Methods

```
>>> my_list = [ 1, 2, 3, 3, 4 ]
>>> my_list.append(5)
>>> my_list
[1, 2, 3, 3, 4, 5]

>>> my_list.extend([6, 7, 8])
>>> my_list
[1, 2, 3, 3, 4, 5, 6, 7, 8]

>>> my_list.insert(2, 15)
>>> my_list.insert(-1, 2)
>>> my_list.append(2)
>>> my_list
[1, 2, 15, 3, 3, 4, 5, 6, 7, 8, 2, 2]

>>> my_list.remove(2)
>>> my_list
[1, 15, 3, 3, 4, 5, 6, 7, 8, 2, 2]
```

```
>>> my_list.pop()       #  2

>>> my_list.index(6)   #  6

>>> my_list.count(2)   #  1

>>> my_list.sort()
>>> my_list
[1, 2, 3, 3, 4, 5, 6, 7, 8, 15]

>>> my_list.reverse()
>>> my_list
[15, 8, 7, 6, 5, 4, 3, 3, 2, 1]

>>> my_list.clear()
>>> my_list
[  ]
```

Work With Code

Iterate Over Lists and Tuples

- my_list = [2, 3, 4, 5]

```
for num in my_list :
        if (num % 2 == 0) :
                print("Even")
        else:
                print("Odd")
```

Test your Knowledge

- 2x = 10 =➔ # x = 10

  Print 2x =➔ # print(x)

- 3 + 5 = y =➔ # y = 3+ 5

  Print (y) =➔ # print (y)

- L = [3, "Hi", 5, 7,.3, [1, 2, 3], 5]

  print (len (L), L[1], L[-1], L[-2])

- For i in range(11): =➔ # for i in range(11) :

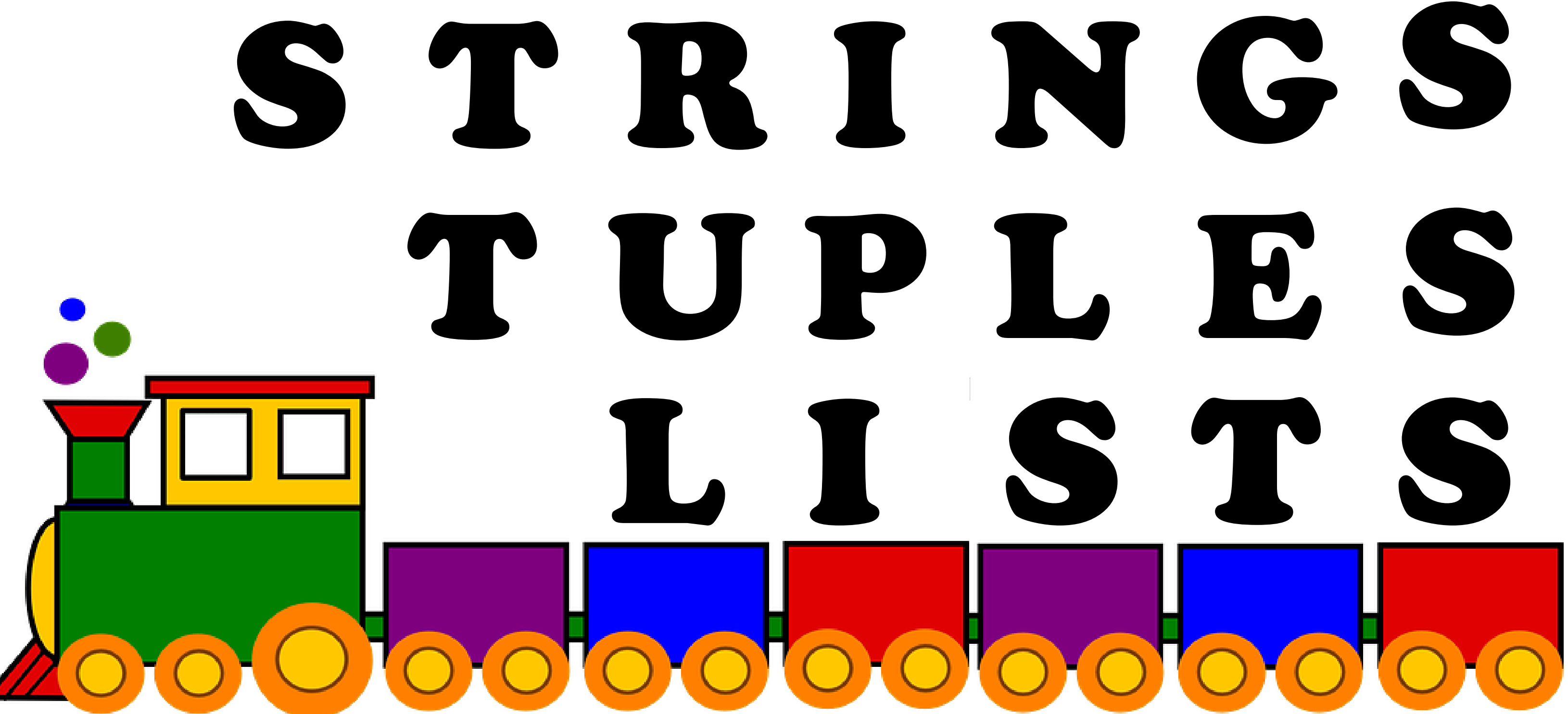
  print ("Hello" * i)

- Use append() to add the string "end" as the last element in L.
- Use del to remove the "end" that you added to the list earlier.
- Use insert() to put the integer 3 after the 7 that you just added to your string

Programming Fundamentals Python

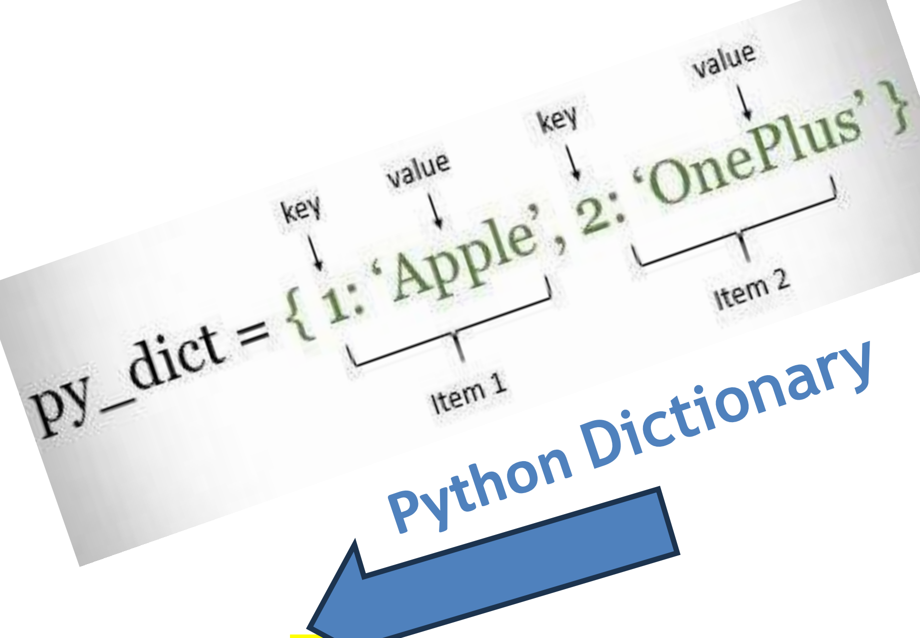
RANGE & CONDITIONAL STATEMENTS

Programming with Python

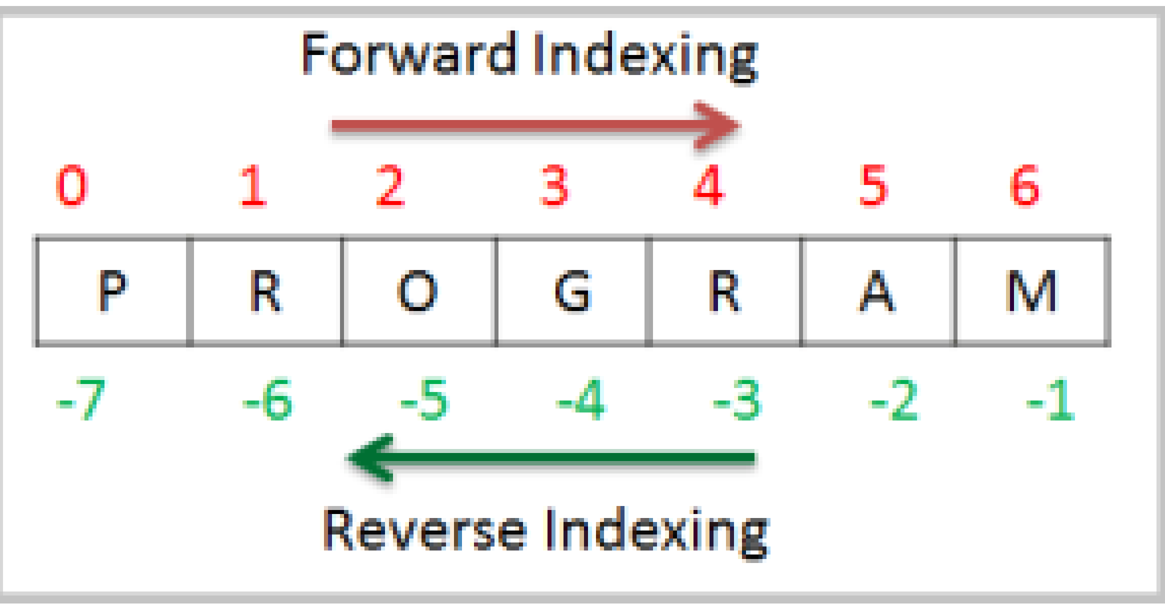


Data Type Wrap Up

- Integers: 2323, 3234
- Floating Point: 32.3, 3.1E2, 65.0
- Complex: 3 + 2j, 1j
- String: ‘text’ , “text”
- Lists: l = [1, 2, 3]
- Tuples: t = (1, 2, 3) or t =1, 2, 3
- Dictionaries: d = { ‘hello’ : ‘there’, 2 : 15 }
- Lists, Tuples, and Dictionaries can store any type (including other lists, tuples, and dictionaries!)
- Only lists and dictionaries are mutable
- All variables are references



STRING



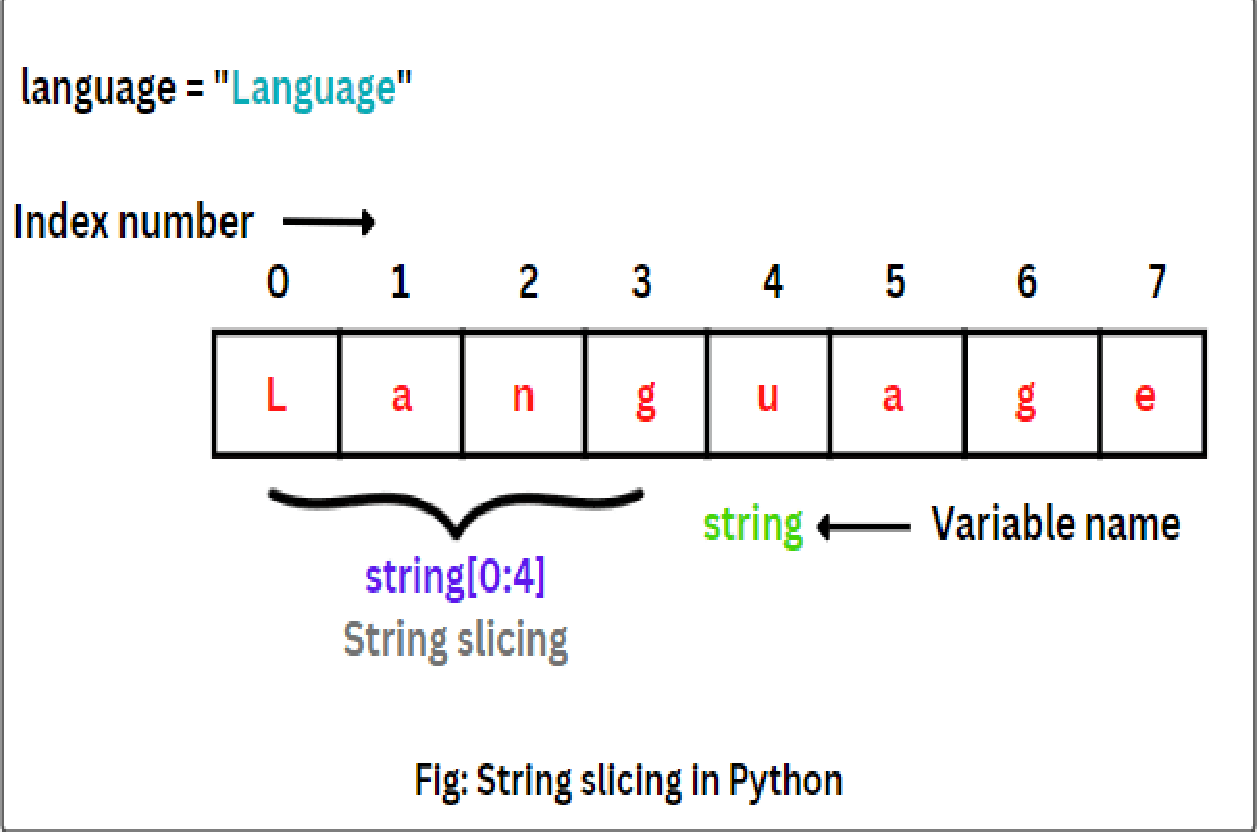


Fig: String slicing in Python

STRING

- Strings in Python have type str .
- They represent sequence of characters .
- Strings are enclosed in single quotes(') or double quotes(“) :
  - Both are equivalent
- Backslash (\) is used to escape quotes and special characters.

```
print (´ What\´s your name? ´ )
print ( “ What´s your name? “ )
```

Concatenate and Repeat

- In Python, + and * operations have special meaning when operating on strings
  - + is used for concatenation of (two) strings
  - * is used to repeat a string, an int number of time

String

• "hello" + "world"	"helloworld"	# concatenation
• "hello" * 3	"hellohellohello"	# repetition
• "hello" [0]	"h"	# indexing
• "hello" [-1]	"o"	# (from end)
• "hello" [1:4]	"ell"	# slicing
• **Len** ("hello")	5	# size
• "hello" < "jello"	True	# comparison
• "e" **in** "hello"	True	# search
• New line:	"escapes: \n "	
• Line continuation:	triple quotes """ """	
• Quotes:	'single quotes', " double quotes "	

Indexing

- Strings can be indexed .
- First character has index 0 .

```
>>> name='Acads'
>>> name[0]
'A'
>>> name[3]
'd'
>>> 'Hello'[1]
'e'
```

Indexing

- Negative indices start counting from the right
- Negatives indices start from -1
- -1 means last, -2 second last, ...

```
>>> name='Acads'
>>> name[-1]
's'
>>> name[-5]
'A'
>>> name[-2]
'd'
```

Indexing

- Using an index that is too large or too small results in "index out of range" error

```
>>> name='Acads'
>>> name[50]

Traceback (most recent call last):
  File "<pyshell#136>", line 1, in <module>
    name[50]
IndexError: string index out of range
>>> name[-50]

Traceback (most recent call last):
  File "<pyshell#137>", line 1, in <module>
    name[-50]
IndexError: string index out of range
```

Slicing

- **To obtain a substring:**
- string[start : end] means substring of string starting at index start and ending at index (end-1)
- string[0 : len(s)] is same as string
- Both start and end are optional
  - If start is omitted, it defaults to 0
  - If end is omitted, it defaults to the length of string
- string[:] is same as string[0 : len(s)], that is same as string.

Slicing

```
>>> name='Acads'
>>> name[0:3]
'Aca'
>>> name[:3]
'Aca'
>>> name[3:]
'ds'
>>> name[:3] + name[3:]
'Acads'
>>> name[0:len(name)]
'Acads'
>>> name[:]
'Acads'
```

More Slicing

```
>>> name='Acads'
>>> name[-4:-1]
'cad'
>>> name[-4:]
'cads'
>>> name[-4:4]
'cad'
```

Understanding Indices for slicing

A	c	a	d	s
0	1	2	3	4
-5	-4	-3	-2	-1

5

Out of Range Slicing

- Out of range indices are ignored for slicing
- when start and end have the same sign, if start >=end, empty slice is returned

A	c	a	d	s
0	1	2	3	4
-5	-4	-3	-2	-1

Why?

```
>>> name='Acads'
>>> name[4:50]
's'
>>> name[40:50]
''
>>> name[-50:20]
'Acads'
```

```
>>> name[-50:-20]
''
>>> name[50:20]
''
>>> name [1:-1]
'cad'
```

Methods in string

- upper()
- lower()
- count(s)   # text.count('word')
- find(s)
- index(s)   # s.index('l')

L I S T S

My_List = [3, 4, 6, 10, 8]

List in Python

- ✓ **Ordered**: Maintain the order of the data insertion.
- ✓ **Changeable**: List is mutable and we can modify items.
- ✓ **Heterogeneous**: List can contain data of different types
- ✓ **Contains duplicate**: Allows duplicates data

Lists

- Ordered sequence of values
- Written as a sequence of comma-separated values between square brackets
- Values can be of different types
  - – usually the items all have the same type

```
lst = [1, 2 ,  3 ,  4, 5]
>>> print(lst)
[1, 2 ,  3 ,  4, 5]
>>> type(lst)
< type ‘ list‘ >
```

Lists

- List is also a sequence type
  - Sequence operations are applicable

```
>>> fib = [1,1,2,3,5,8,13,21,34,55]
>>> len(fib)
10
>>> fib[3] # Indexing
3
>>> fib[3:] # Slicing
[3, 5, 8, 13, 21, 34, 55]
```

Lists

- List is also a sequence type
  - Sequence operations are applicable

```
>>> [0] + fib # Concatenation
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
>>> 3 * [1, 1, 2] # Repetition
[1, 1, 2, 1, 1, 2, 1, 1, 2]
>>> x,y,z = [1,1,2] #Unpacking
>>> print (x, y, z )
1 1 2
```

Lists

List = [1, 2, 3, 4, 5, "a", "b", "c", 3.4, 2.4, 2.6]

Nested_list = [8, [1, 2, 3], [4, 5, 6], 7, 9] # Nested List

List Length # len(List)

Update a Value in a List # list[0] = ‘H’

Add a Value to a List # list.append(10)

list.insert (5, 6)

More Operations on Lists

- L.append(x)
- L.extend(seq)
- L.insert(i, x)
- L.remove(x)
- L.pop(i)
- L.pop()
- L.index(x)
- L.count(x)
- L.**sort**()
- L.**reverse**()

x is any value, L is a sequence value (list) and i is an integer value.

List Methods

```
>>> my_list = [ 1, 2, 3, 3, 4 ]
>>> my_list.append(5)
>>> my_list
[1, 2, 3, 3, 4, 5]

>>> my_list.extend([6, 7, 8])
>>> my_list
[1, 2, 3, 3, 4, 5, 6, 7, 8]

>>> my_list.insert(2, 15)
>>> my_list.insert(-1, 2)
>>> my_list.append(2)
>>> my_list
[1, 2, 15, 3, 3, 4, 5, 6, 7, 8, 2, 2]

>>> my_list.remove(2)
>>> my_list
[1, 15, 3, 3, 4, 5, 6, 7, 8, 2, 2]
```

```
>>> my_list.pop()      #  2

>>> my_list.index(6)   #  6

>>> my_list.count(2)   #  1

>>> my_list.sort()
>>> my_list
[1, 2, 3, 3, 4, 5, 6, 7, 8, 15]

>>> my_list.reverse()
>>> my_list
[15, 8, 7, 6, 5, 4, 3, 3, 2, 1]

>>> my_list.clear()
>>> my_list
[  ]
```

Mutable and Immutable Types

- Tuples and List types look very similar
- However, there is one major difference: Lists are ***mutable***
  - Contents of a list can be modified
- Tuples and Strings are ***immutable***
  - Contents can not be modified

TUPLES

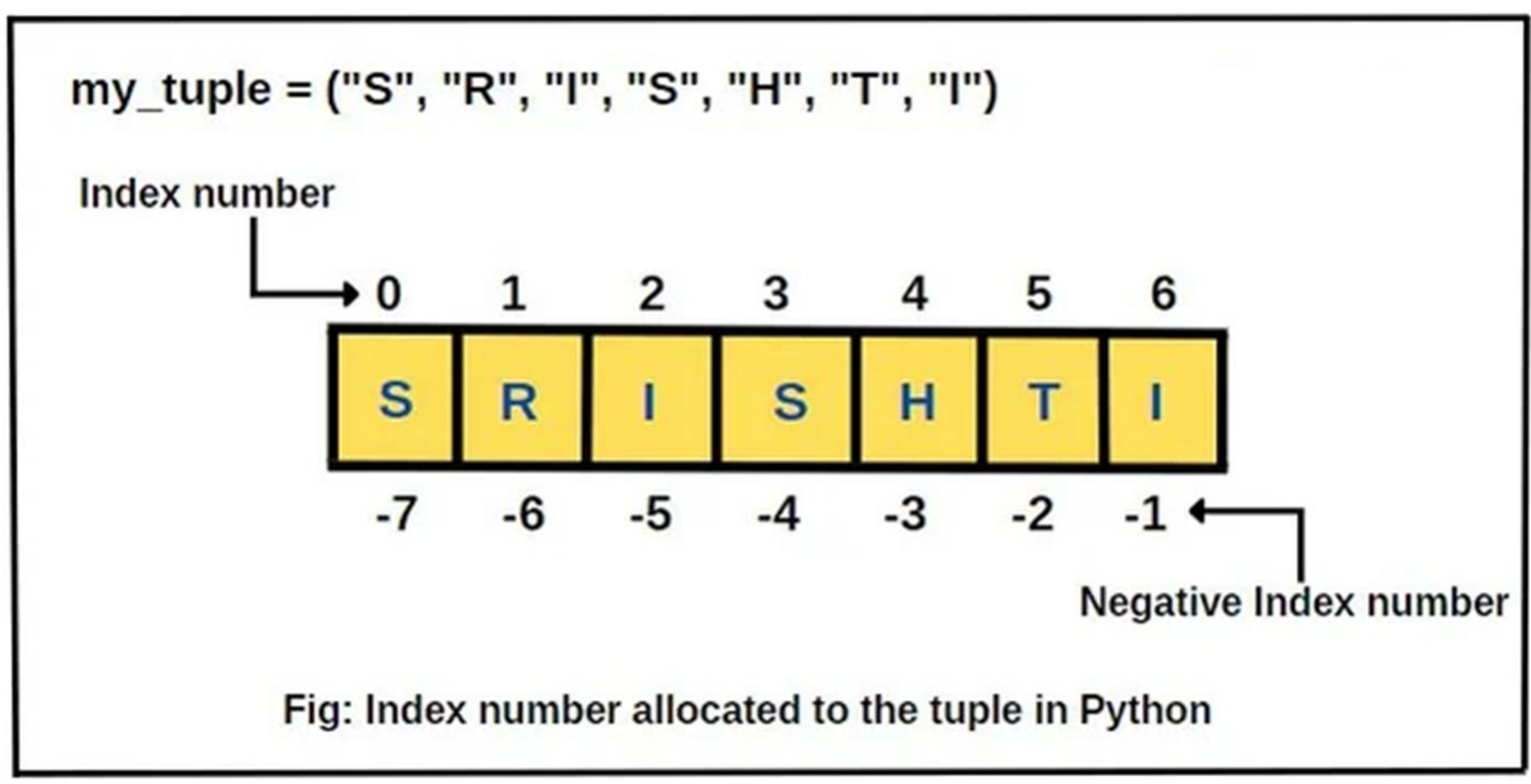


Fig: Index number allocated to the tuple in Python

WHAT IS a TUPLE?

A tuple: is a sequence of values, which can be of any type and they are indexed by integer. Tuples are just like list, but we can't change values of tuples in place. Thus tuples are ***immutable***.

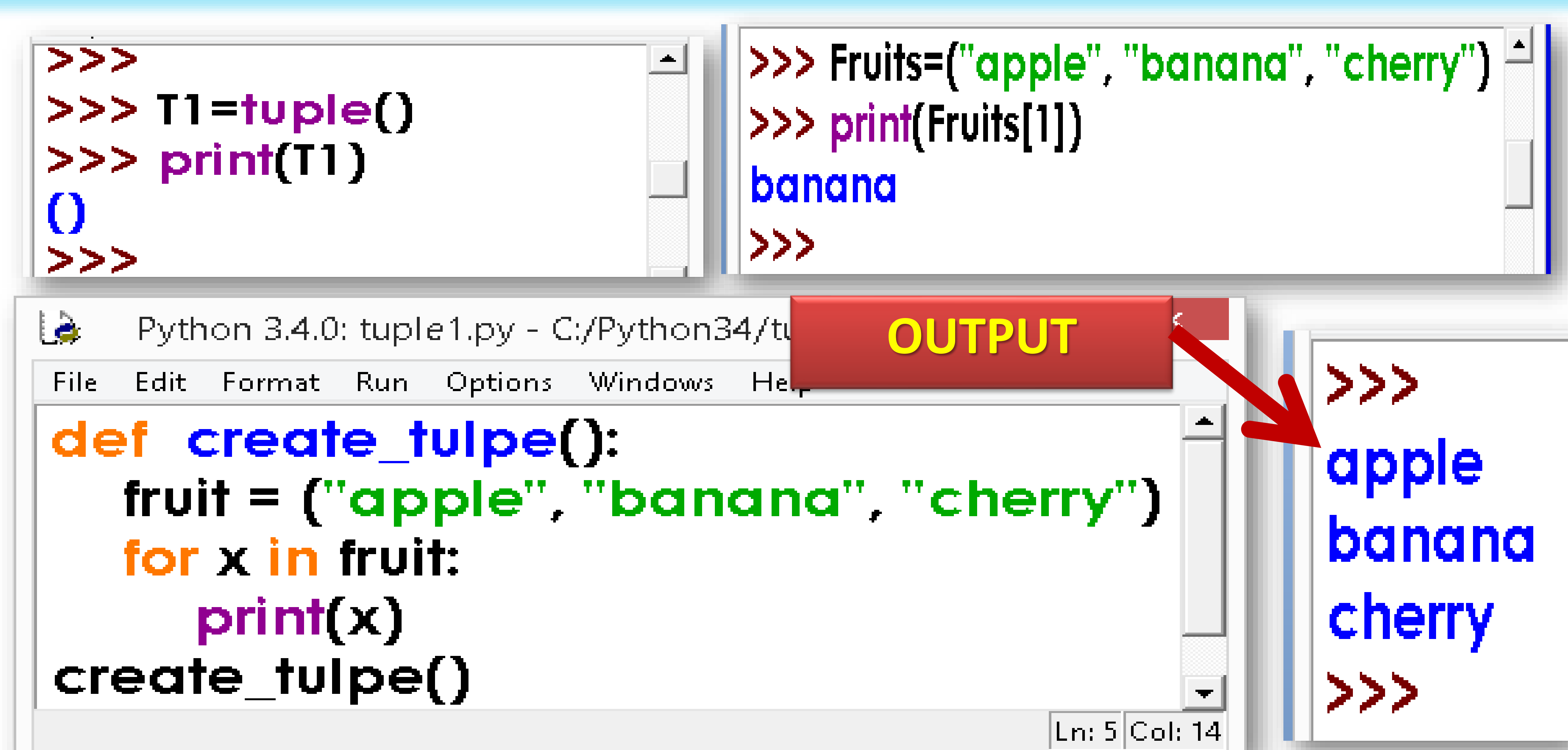
The index value of tuple starts from 0.

A tuple consists of a number of values separated by commas.

For example:

```
>>> T=10, 20, 30, 40
>>> print (T)
(10, 20, 30, 40)
```

CREATING TUPLE And ACCESSING to it



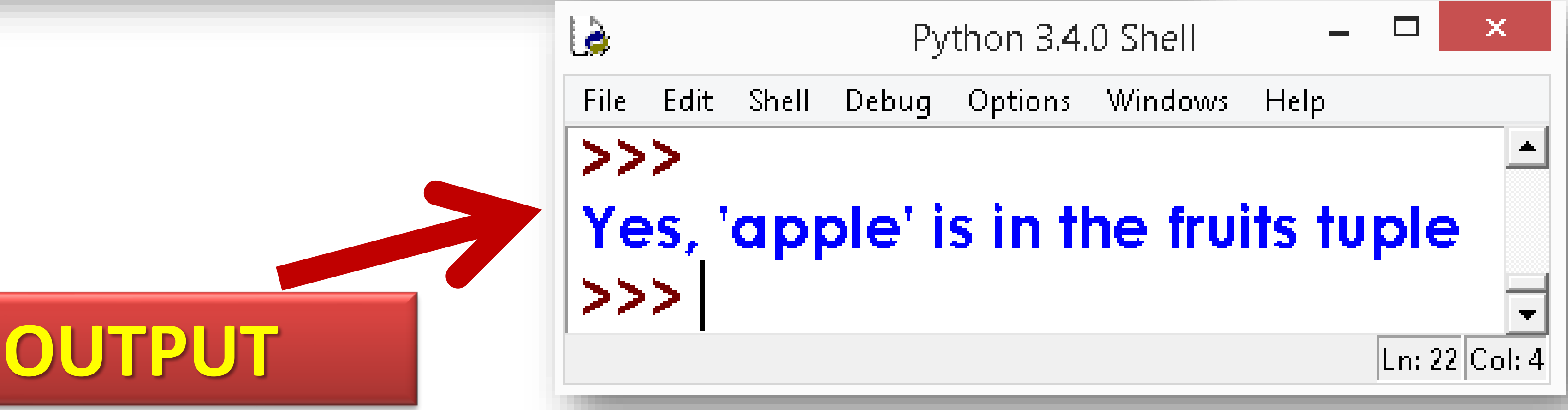
CHECK IF ITEM EXISTS

Python 3.4.0: tuple2.py - C:/Python34/tuple2.py

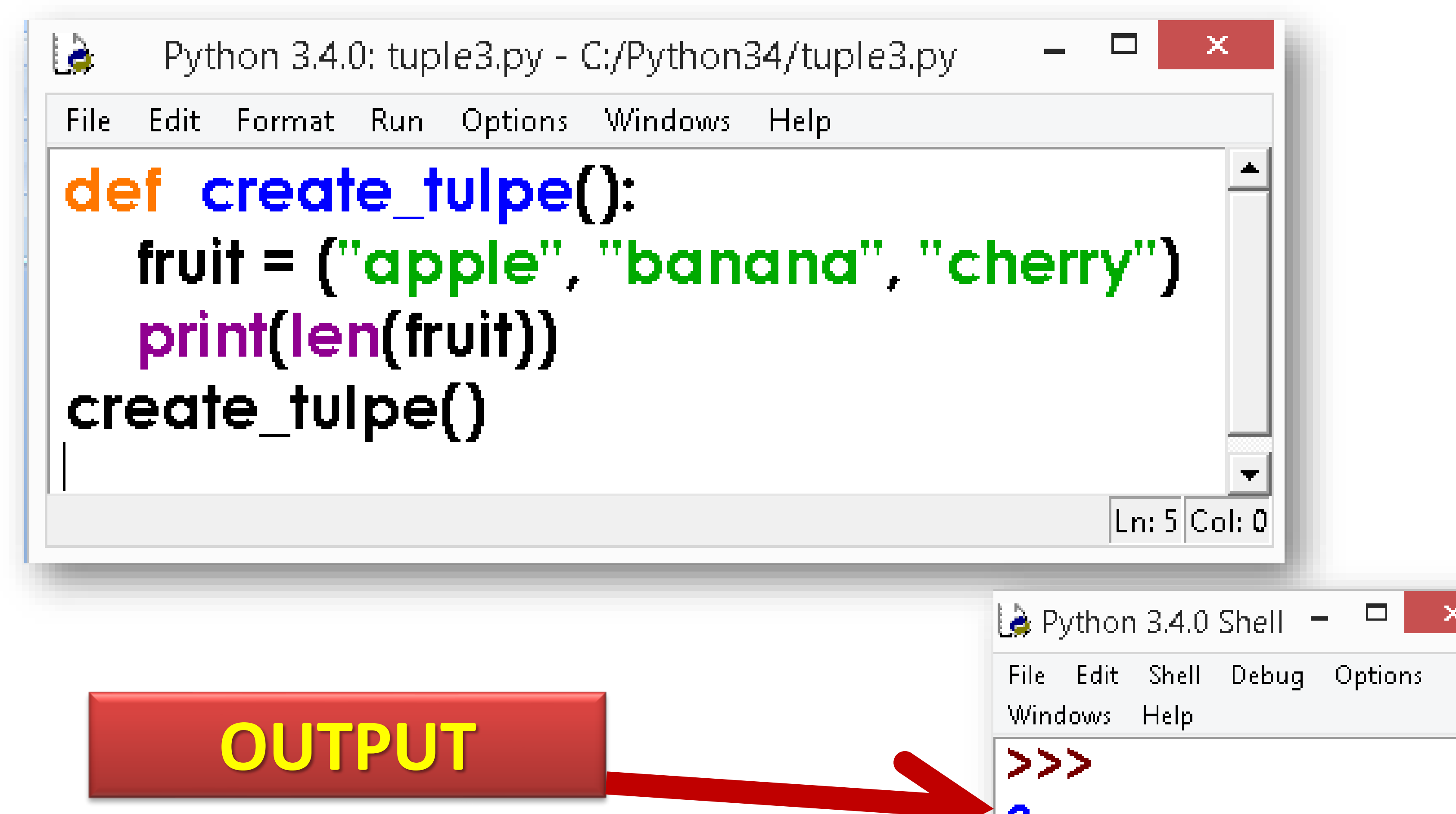
File Edit Format Run Options Windows Help

```
def  create_tulpe():
    fruit = ("apple", "banana", "cherry")
    if "apple" in fruit:
        print("Yes, 'apple' is in the fruits tuple"
create_tulpe()
```

Ln: 6 Col: 0



TUPLE LENGTH



REMOVING A TUPLE

You cannot remove or delete or update items in a tuple.

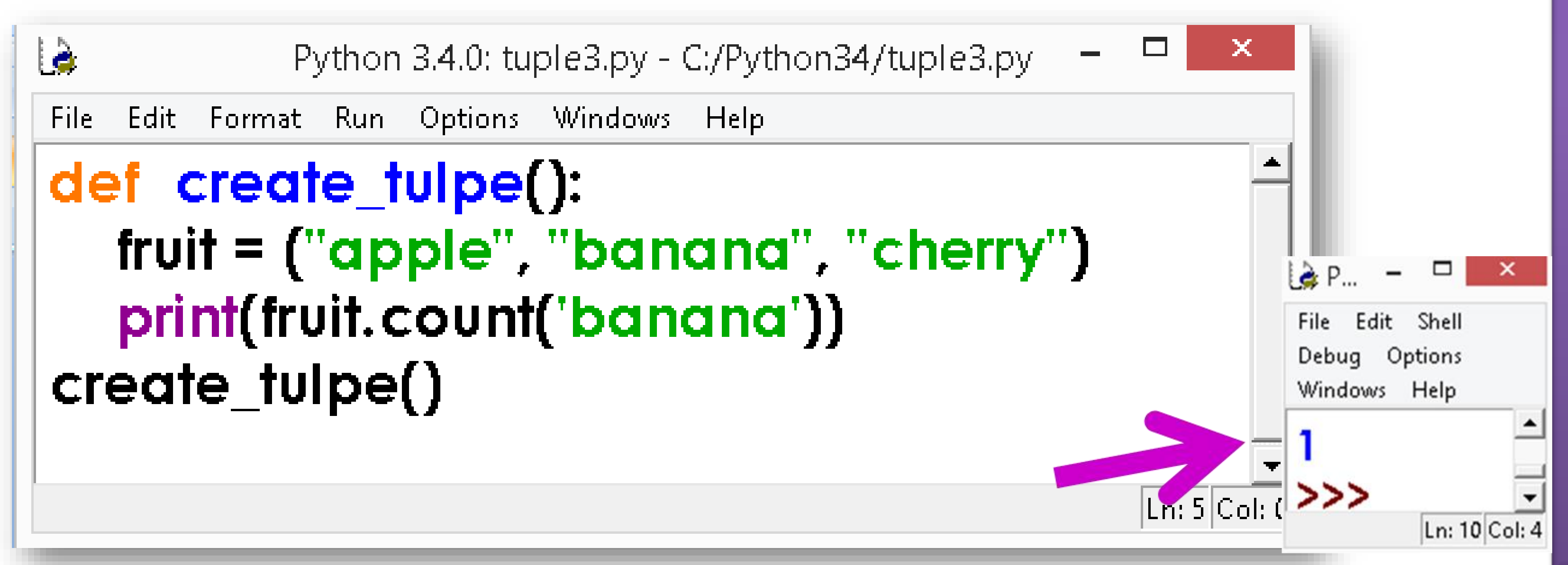
Tuples are unchangeable, so you cannot remove items from it, but you can delete the tuple completely:

NOTE: TUPLES ARE IMMUTABLE

TUPLE METHODS (1)

1. count() Method

Return the number of times the value appears in the tuple



Count() method returns total times of 'banana' present in the given tuple

TUPLE METHODS (2)

2. index() Method

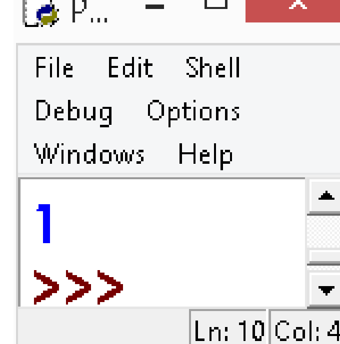
Python 3.4.0: tuple5.py - C:/Python34/tuple5.py

File Edit Format Run Options Windows Help

```
def  create_tulpe():
    fruit = ("apple", "banana", "cherry")
    print(fruit.index("banana"))
create_tulpe()
```

Ln: 5 Col: 0

index() Method



index() Method returns index of "banana" i.e 1

More Operations on Tuples

- Tuples can be concatenated, repeated, indexed and sliced

Course1 = (´Python´, Ámey´, 101)

Course2 = (´Stats´, Ádams´, 102)

```
>>> course1
('Python', 'Amey', 101)
>>> course2
('Stats', 'Adams', 102)
>>> course1 + course2
('Python', 'Amey', 101, 'Stats', 'Adams', 102)
>>> (course1 + course2)[3]
'Stats'
>>> (course1 + course2)[2:7]
(101, 'Stats', 'Adams', 102)
>>> 2*course1
('Python', 'Amey', 101, 'Python', 'Amey', 101)
```

Tuples and Assignment

- We can also put a tuple on the left-hand side of an assignment statement
- We can even omit the parentheses

```
>>> (x, y) = (4, 'fred')
>>> print(y)
fred
>>> (a, b) = (99, 98)
>>> print(a)
99
```

but... Tuples are "immutable"

Unlike a list, once you create a tuple, you cannot alter its contents - similar to a string

```
>>> x = [9, 8, 7]
>>> x[2] = 6
>>> print(x)
>>>[9, 8, 6]
>>>
```

```
>>> y = 'ABC'
>>> y[2] = 'D'
Traceback:'str' object does
not support item
Assignment
>>>
```

```
>>> z = (5, 4, 3)
>>> z[2] = 0
Traceback:'tuple'
object does
not support item
Assignment
>>>
```

Summary of Sequences

Operation	Meaning
seq[i]	i-th element of the sequence
len(seq)	Length of the sequence
seq1 + seq2	Concatenate the two sequences
num*seq seq*num	Repeat seq num times
seq[start:end]	slice starting from **start**, and ending at **end-1**
e in seq	True if e is present in seq, False otherwise
e not in seq	True if e is not present in seq, False otherwise
for e in seq	Iterate over all elements in seq (e is bound to one element per iteration)

Sequence types include String, Tuple and List.
Lists are mutable, Tuple and Strings immutable.

DIFFERENCE BETWEEN LIST AND TUPLE

LIST	TUPLE
Syntax for list is slightly different comparing with tuple.	Syntax for tuple is slightly different comparing with lists
Weekdays=[‘Sun’,’Mon’, ‘wed’,46,67] type(Weekdays) # square brackets [ ] class<‘lists’>	twdays = (‘Sun’, ‘mon', ‘tue', 634) type(twdays) # rounded brackets ( ) class<‘tuple’>
List can be edited once it is created in python. Lists are mutable data structure.	A tuple is a list which one cannot edit once it is created in Python code. The tuple is an immutable data structure
More methods or functions are associated with lists.	Compare to lists tuples have Less methods or functions.

TUPLE and List METHODS

```
>>> l = list()
>>> dir(l)
['append', 'count', 'extend', 'index', 'insert',
'pop', 'remove', 'reverse', 'sort']

>>> t = tuple()
>>> dir(t)
['count', 'index']
```

RANGE

RANGE

- range(From, TO, step) =➔ (S, E, d)
  - generates the list:
    - [S, S + d, S + 2*d, ..., S + k*d]
    - where $S + k*d < E <= S + (K + 1)*d$
- range(S, E) is equivalent to range(S, E, 1)
- range(E) is equivalent to range(0, E)

Exercise: What if d is negative? Use python interpreter to find out.

The range() Function

- Create an empty list.

```
new_list = []
```

- Use the range() and append() functions to add the integers 1-20 to the empty list.

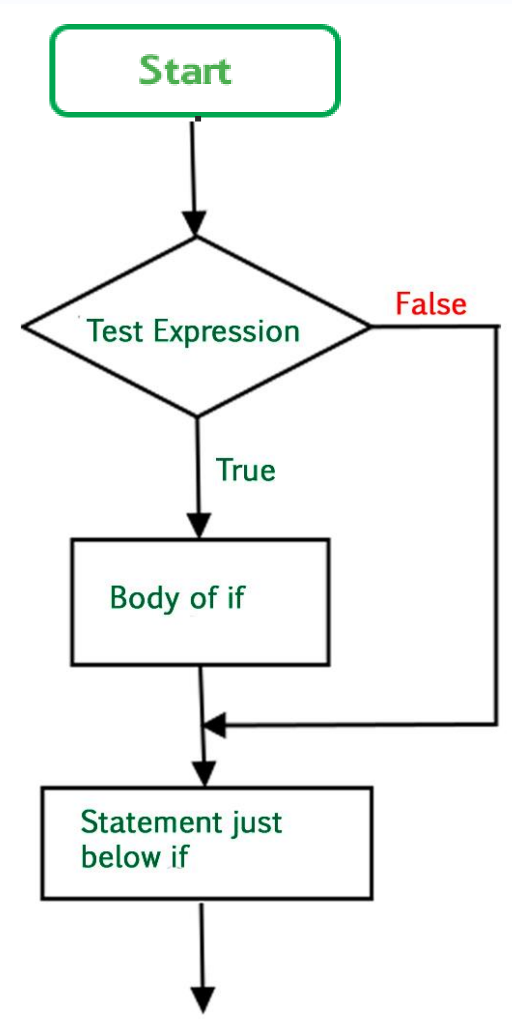
```
for i in range(1, 21):
    new_list.append(i)
```

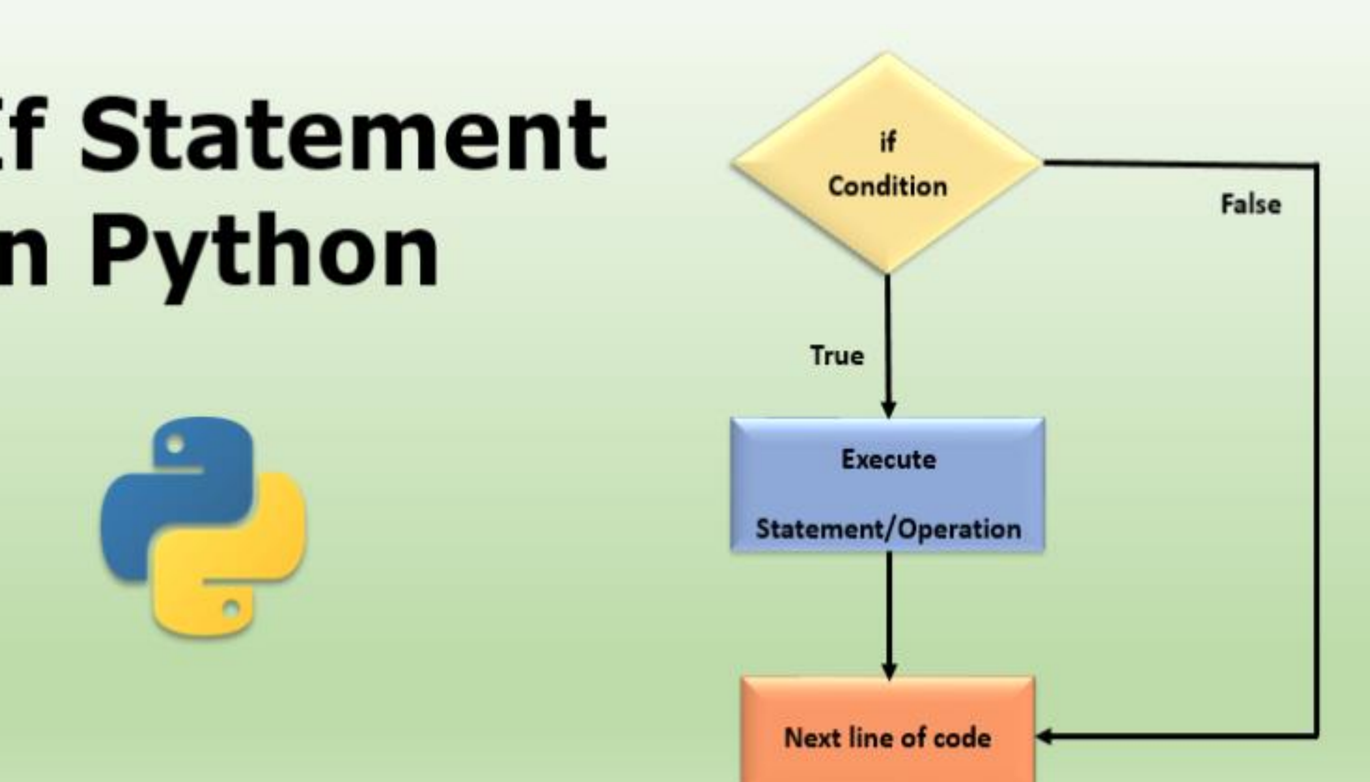
- Print the list to the screen, what do you have?

```
print(new_list)
```

Output: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]

Conditional Statements in Python





Conditional Statements

- In daily routine
  - If it is very hot, I will skip exercise.
  - If there is a quiz tomorrow, I will first study and then sleep. Otherwise, I will sleep now.
  - If I have to buy coffee, I will go left. Else I will go straight.

If Statement

In Python, the if statement is used for conditional branching. It allows you to execute a block of code only if a certain condition is true.

```
if condition :
    # execute this block of code
else :
    # execute this block of code
```

if-else Statement

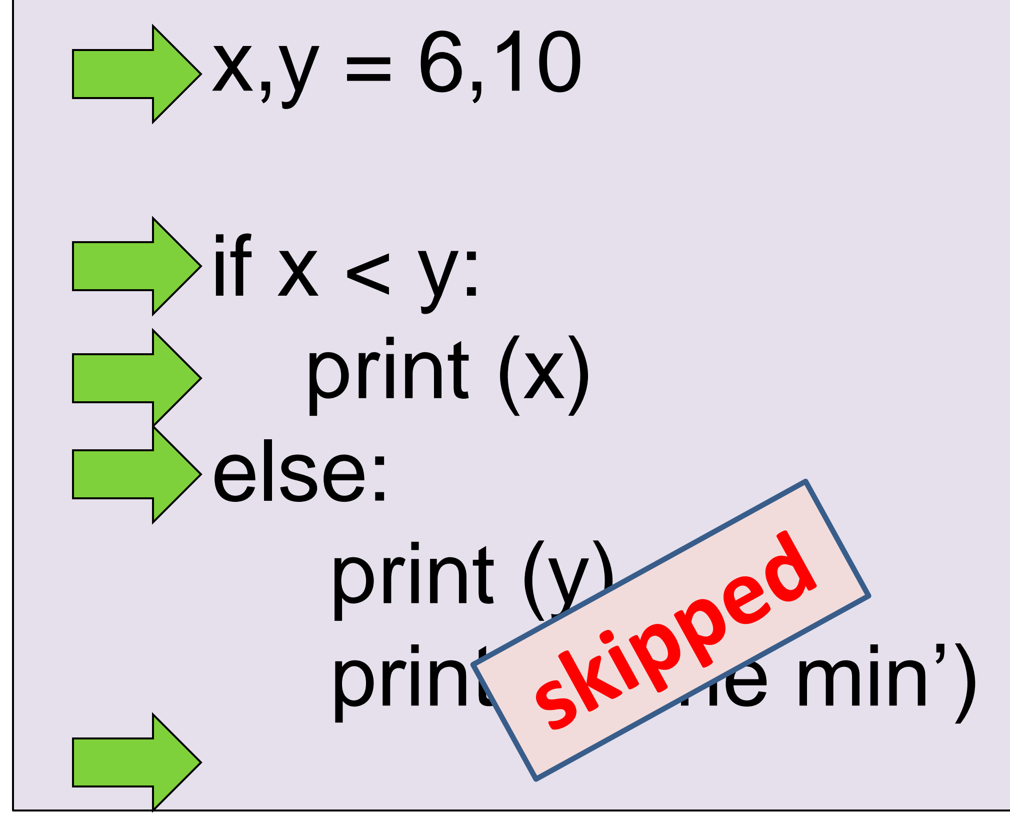
- Compare two integers and print the min.

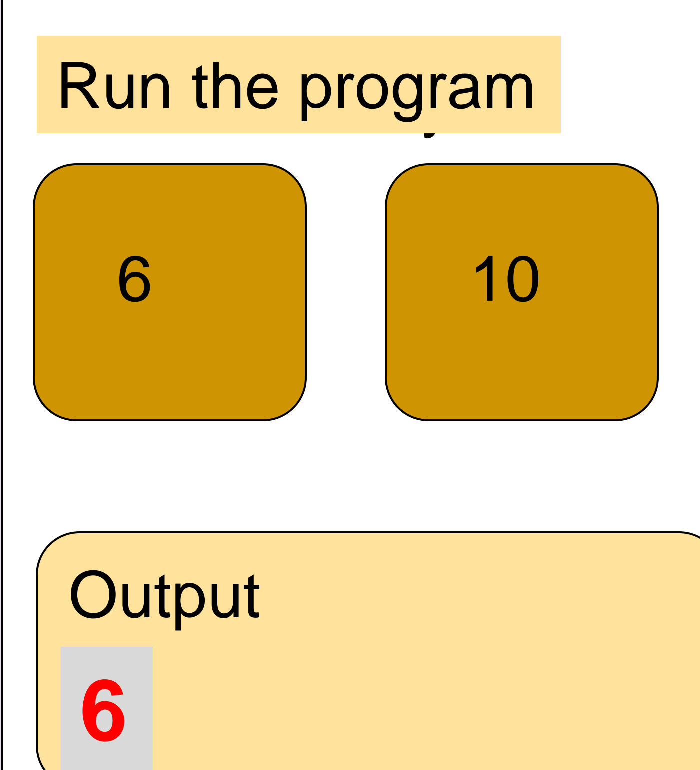
```
if  x < y:
        print (x)
else:
        print (y)
print ('is the minimum')
```

1. Check if x is less than y.
2. If so, print x
3. Otherwise, print y.

Indentation

- Indentation is **important** in Python
  - grouping of statement (block of statements)
  - no explicit brackets, e.g. { }, to group statements

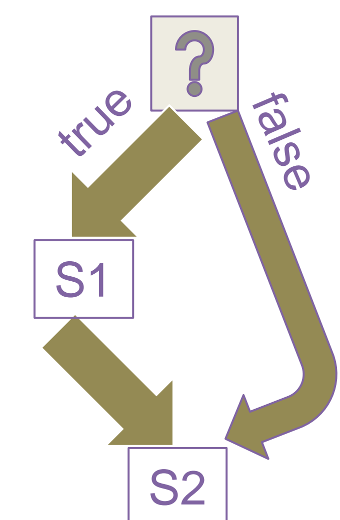




if Statement (no else!)

- General form of the if statement

```
if boolean-expr :
        S1
S2
```

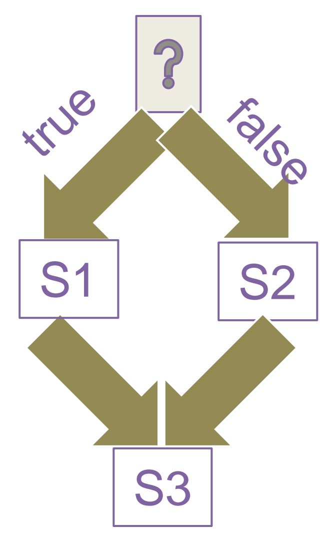


- Execution of if statement
  - First the expression is evaluated.
  - If it evaluates to a **true** value, then S1 is executed and then control moves to the S2.
  - If expression evaluates to **false**, then control moves to the S2 directly.

if-else Statement

- General form of the if-else statement

```
if boolean-expr :
        S1
else:
        S2
S3
```



- Execution of if-else statement
  - First the expression is evaluated.
  - If it evaluates to a **true** value, then S1 is executed and then control moves to S3.
  - If expression evaluates to **false**, then S2 is executed and then control moves to S3.
  - S1/S2 can be **blocks** of statements!

Nested if, if-else

```
if a <= b:
	if a <= c:
		…
	else:
		…
else:
	if b <= c) :
		…
	else:
		…
```

elif

- A special kind of nesting is the chain of if-else-if-else-… statements
- Can be written elegantly using if-elif-..-else

```
if cond1:
        s1
else:
    if cond2:
        s2
    else:
        if cond3:
            s3
        else:
            …
```

```
if cond1:
        s1
elif cond2:
    s2
elif cond3:
    s3
elif  …
else
     last-block-of-
stmt
```

Summary of if, if-else

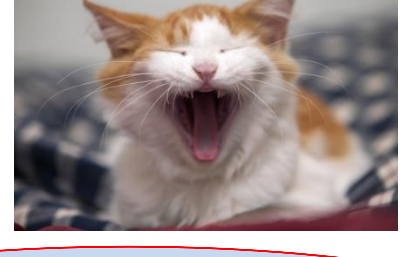
- if-else, nested if's, elif.
- Multiple ways to solve a problem
  - issues of readability, maintainability and efficiency

Quiz

- What is the value of expression:

(5<2) and (3/0 > 1)

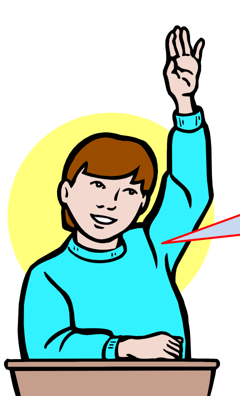
a) Run time crash/error



b) I don’t know / I don’t care

c) False

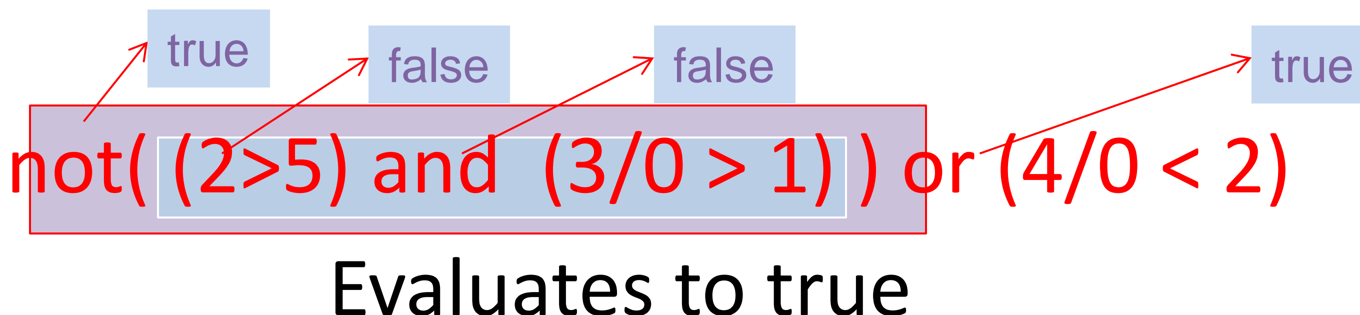
d) True



The correct answer is **False**

Short-Circuit Evaluation

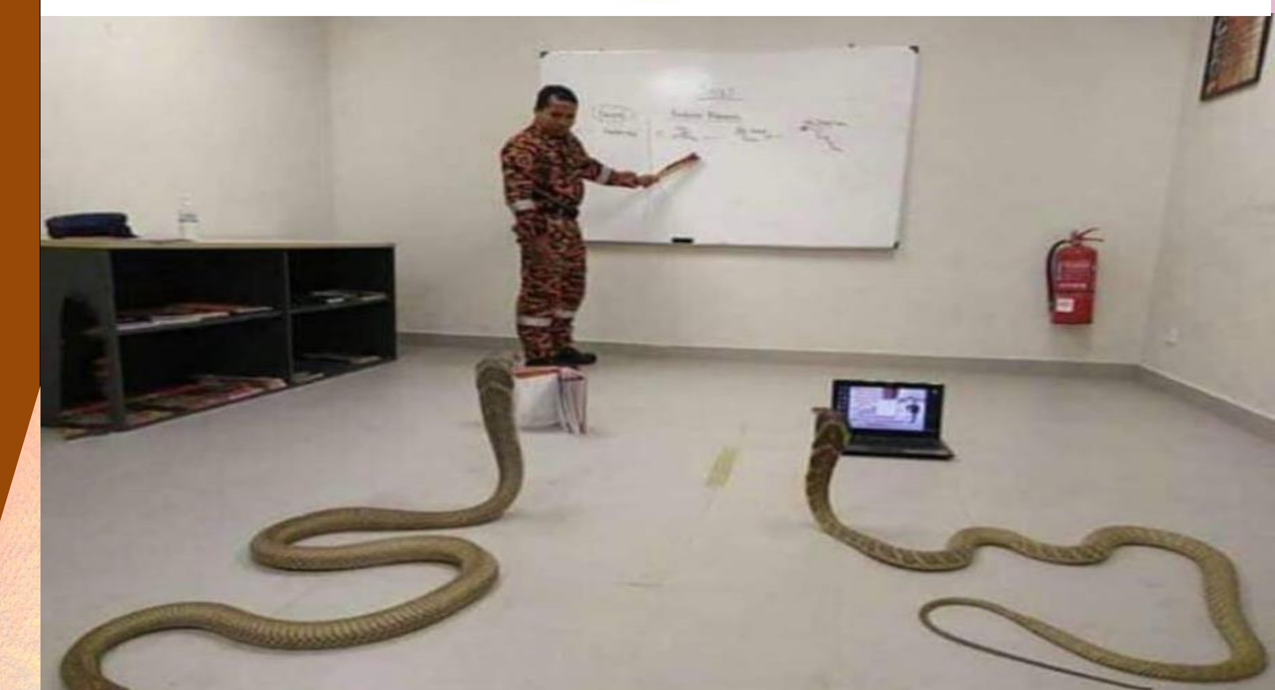
- Do not evaluate the second operand of binary short-circuit logical operator if the result can be deduced from the first operand
  - Also applies to nested logical operators



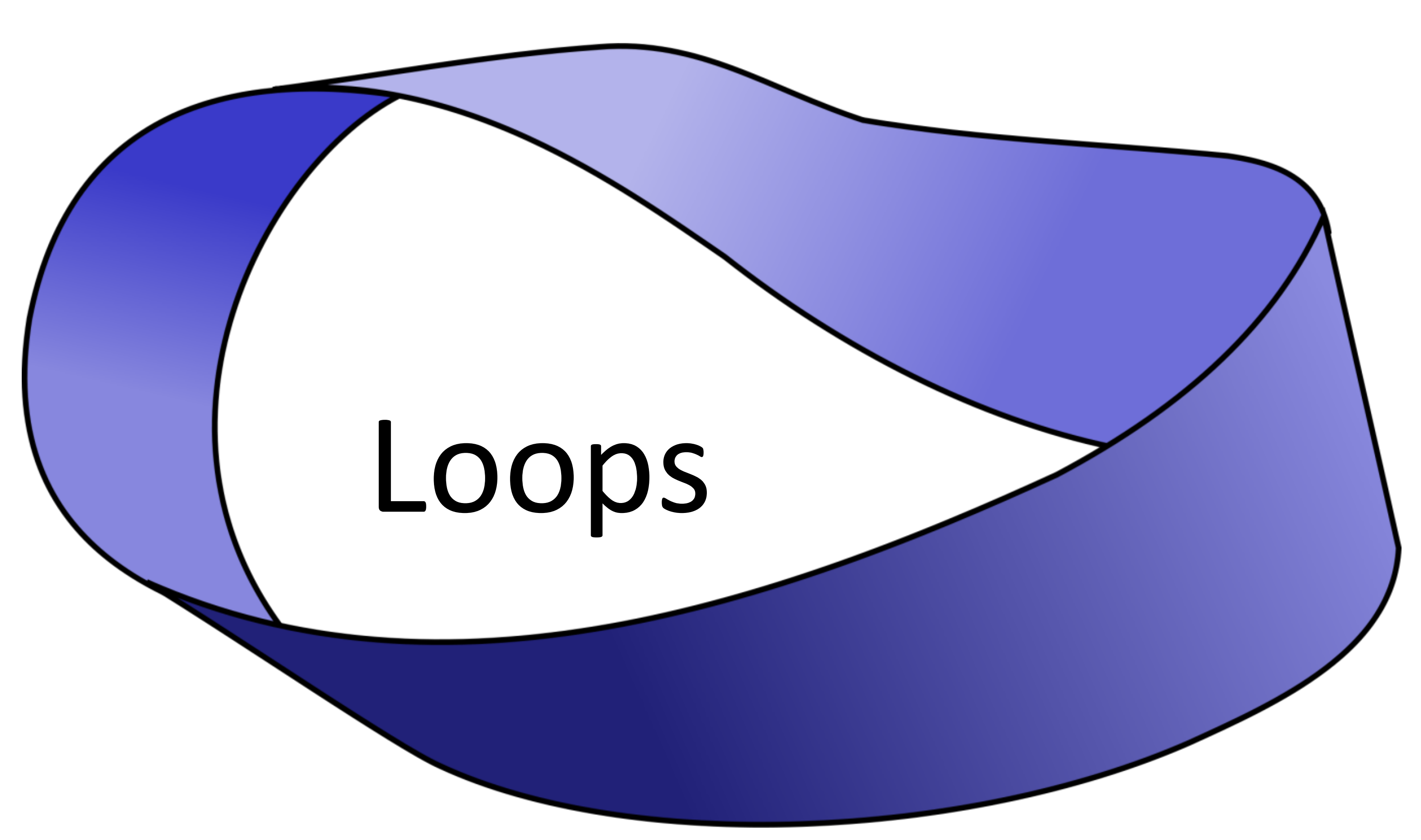
Evaluates to true

Programming Fundamentals Python

Looping Statements



Programming using Python



Loops

- Loops make a section of the program to be repeated a certain number of times.
- Repeats until the condition remains true.
- Terminates when the condition becomes false.

Loops in Python

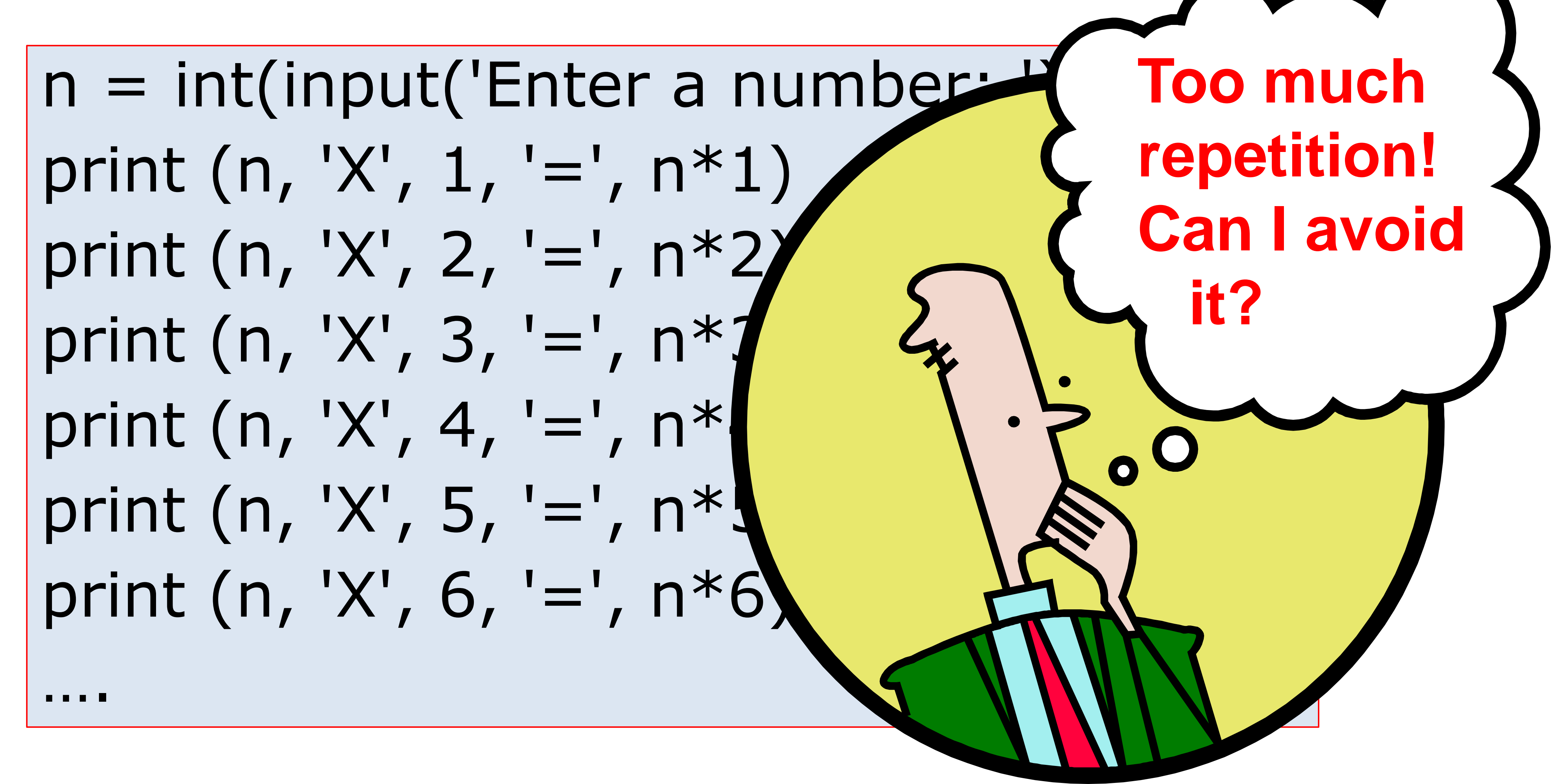
There are two types of loops built into Python:

1. for loop
2. while loop

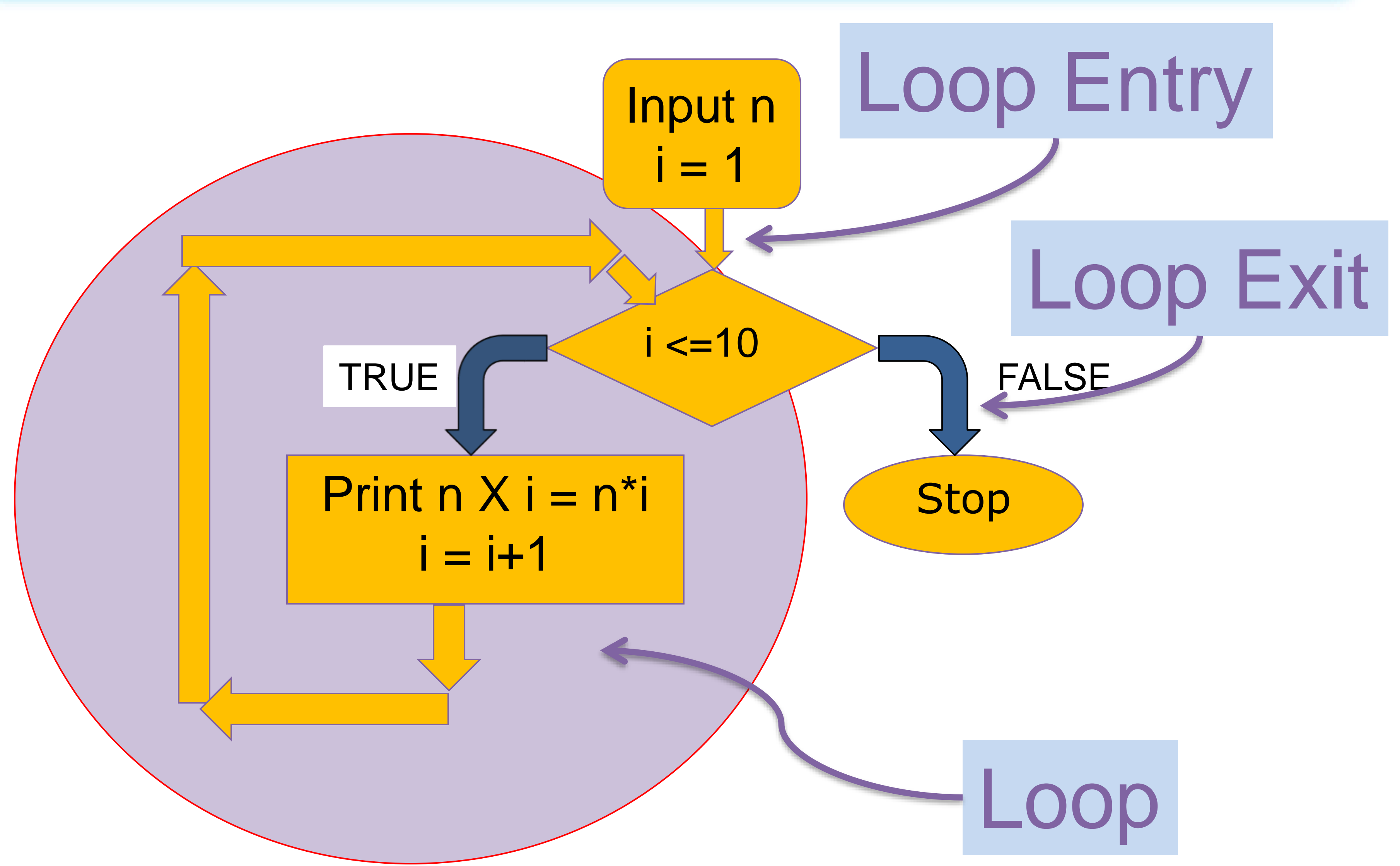
Printing Multiplication Table

5	X	1	=	5
5	X	2	=	10
5	X	3	=	15
5	X	4	=	20
5	X	5	=	25
5	X	6	=	30
5	X	7	=	35
5	X	8	=	40
5	X	9	=	45
5	X	10	=	50

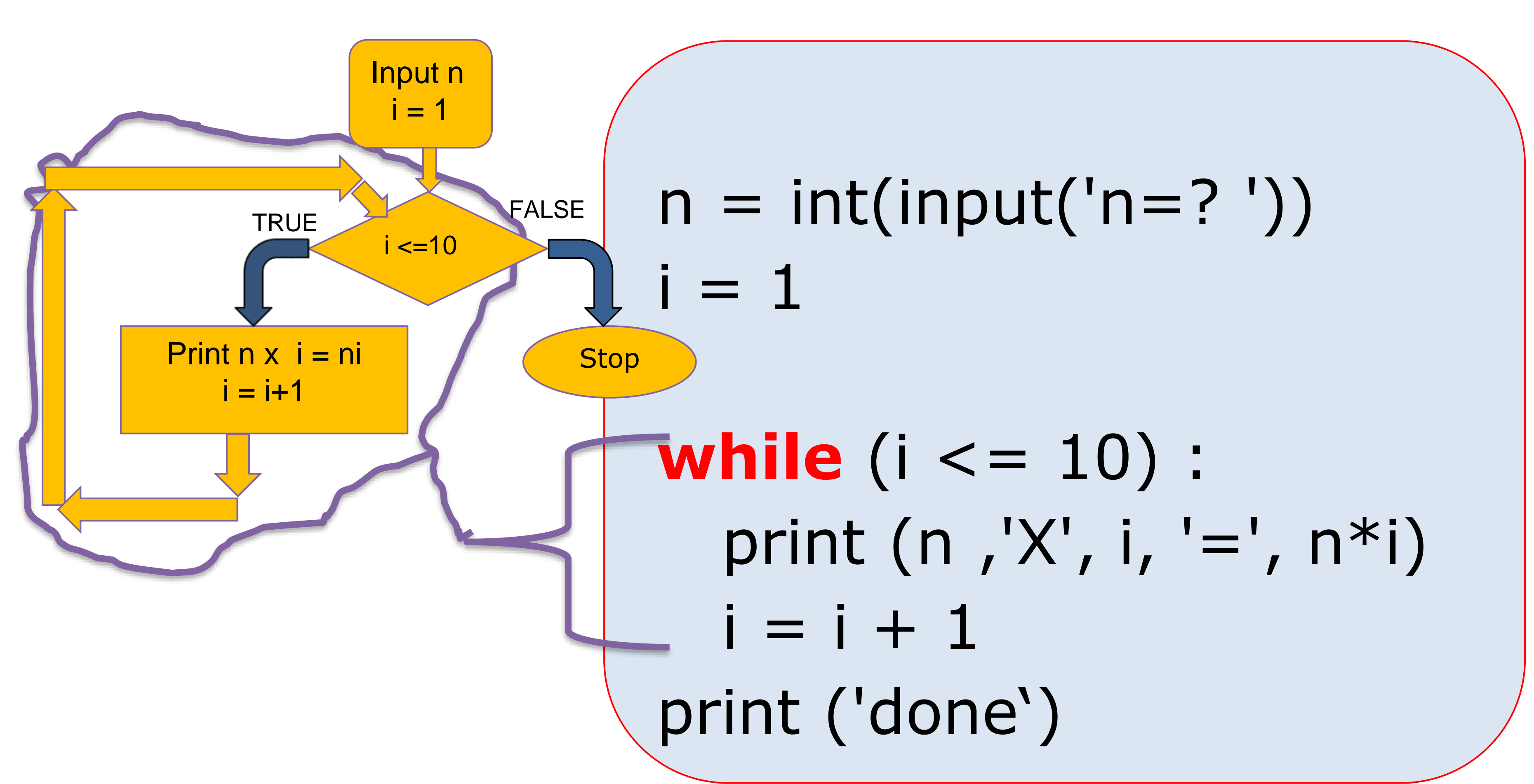
Program...



Printing Multiplication Table

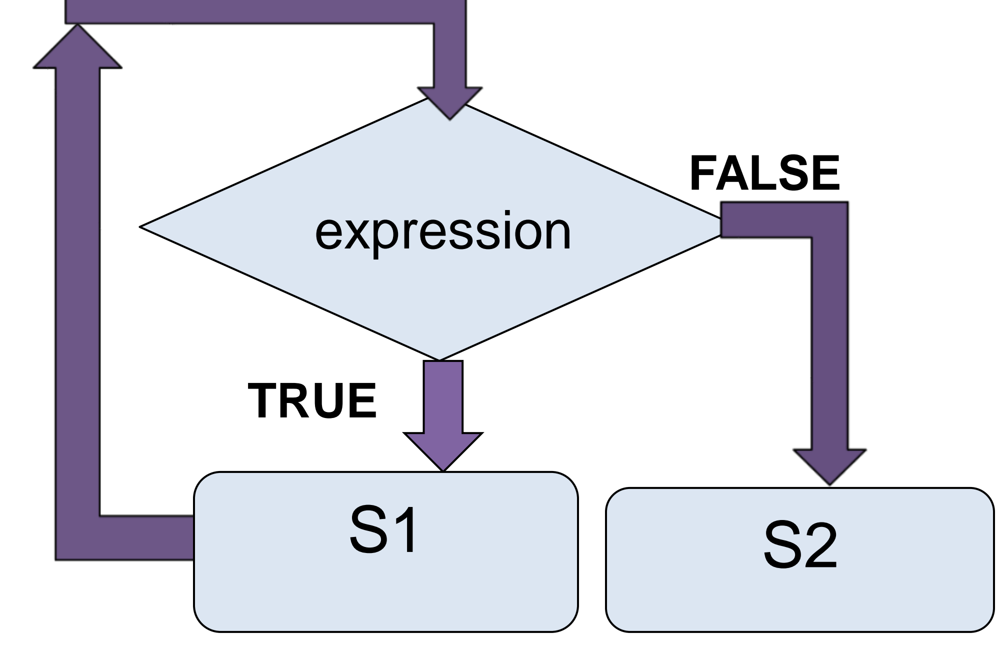


Printing Multiplication Table



While Statement

```
while (expression):
        S1
S2
```



1. Evaluate expression
2. If TRUE then
   a) execute statement1 (S1)
   b) goto step 1.
3. If FALSE then execute statement2 (S2).

Quiz

- What will be the output of the following program

```
# print all odd numbers < 10
i = 1
while i <= 10:
    if i%2==0: # even
        continue
    print (i, end=' ')
    i = i+1
```

Continue and Update Expr

- Make sure continue does not by pass update-expression for while loops

```
# print all odd numbers < 10
i = 1
while i <= 10:
    if i%2==0: # even
        continue
    print (i, end=' ')
    i = i+1
```

i is not incremented when even number encountered. Infinite loop!!

Calculate the Sum of Numbers

```
# program to calculate the sum of numbers
# until the user enters zero
total = 0
number = int(input('Enter a number: '))
# add numbers until number is zero
while number != 0:
    total += number    # total = total + number
    # take integer input again
    number = int(input('Enter a number: ‘))

print('total =', total)
```

For Loop

- Print the sum of the reciprocals of the first 100 natural numbers.

```
rsum = 0.0# the reciprocal sum

# the for loop
for i in range(1,101):
    rsum = rsum + 1.0/i
print ('sum is', rsum)
```

For loop in Python

- General form:

```
for variable in sequence:
    stmt
```

- **For loops essentially say:**

"For all elements in a sequence, do something"

Or: "Repeats a set of statements over a group of values".

for loop - Syntax

```
for j in range(5) :
        print (j * 2)
        print (j * j)
        print (j * j*j)
```

(for loop) -- Exercise

- Get a number form user and calculate its factorial.

```
Fact = 1
for i in range( 1, n + 1):
      fact = fact * I          # We have to replace I with i and fact with Fact
      Fact = Fact * i
print ("factorial of ", n,  "=" , fact )
```

?

(for loop) -- Exercise-2

- Write a program that ask the user to enter a number. The program should print the **Cube** of all integers starting from 1 to the Number.

E.g.,

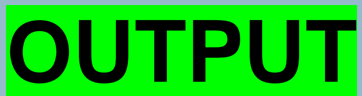
```
Enter a Number: 4
1    1
2    8
3    27
4    64
```

Example of Repetition

```
n = int ( input(“Enter the value n”) )
for  i in range (1, n+1):

     print (i, “ Potato”  )
```

Example of Repetition

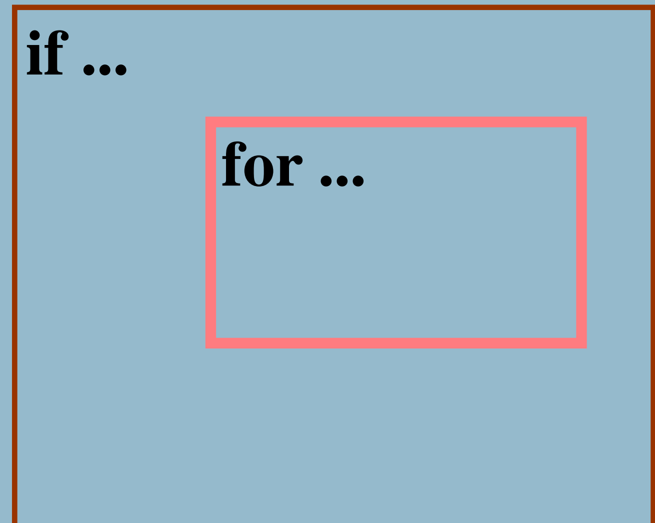
```
n = int ( input("Enter the value n") )        # n = 3
for  num  in range (1, n+ 1) :

     print (num ,  " Potato" )
```

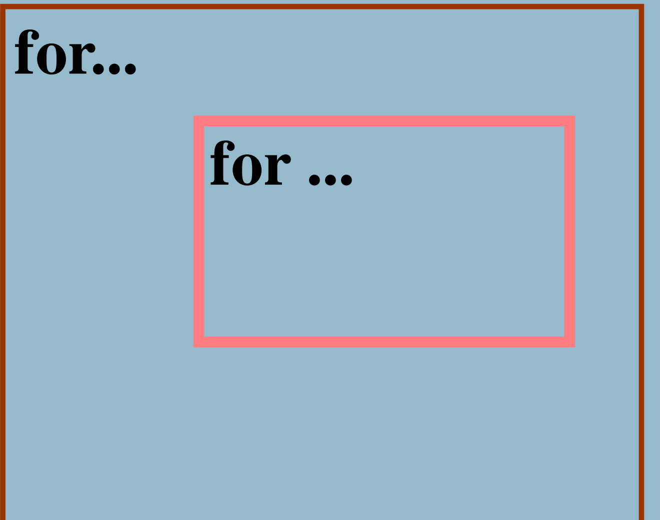



```
Enter the value n
1Potato
2Potato
3Potato
```

Nested Loops

Recall when a control structure is contained within another control structure, the inner one is said to be *nested*.





You may have repetition within decision and vice versa.

Nesting of for loop – EX.

// Program to print a specific Pattern

```
for i in range (1,6):
    print ("*" * i)
print ("\n")
```

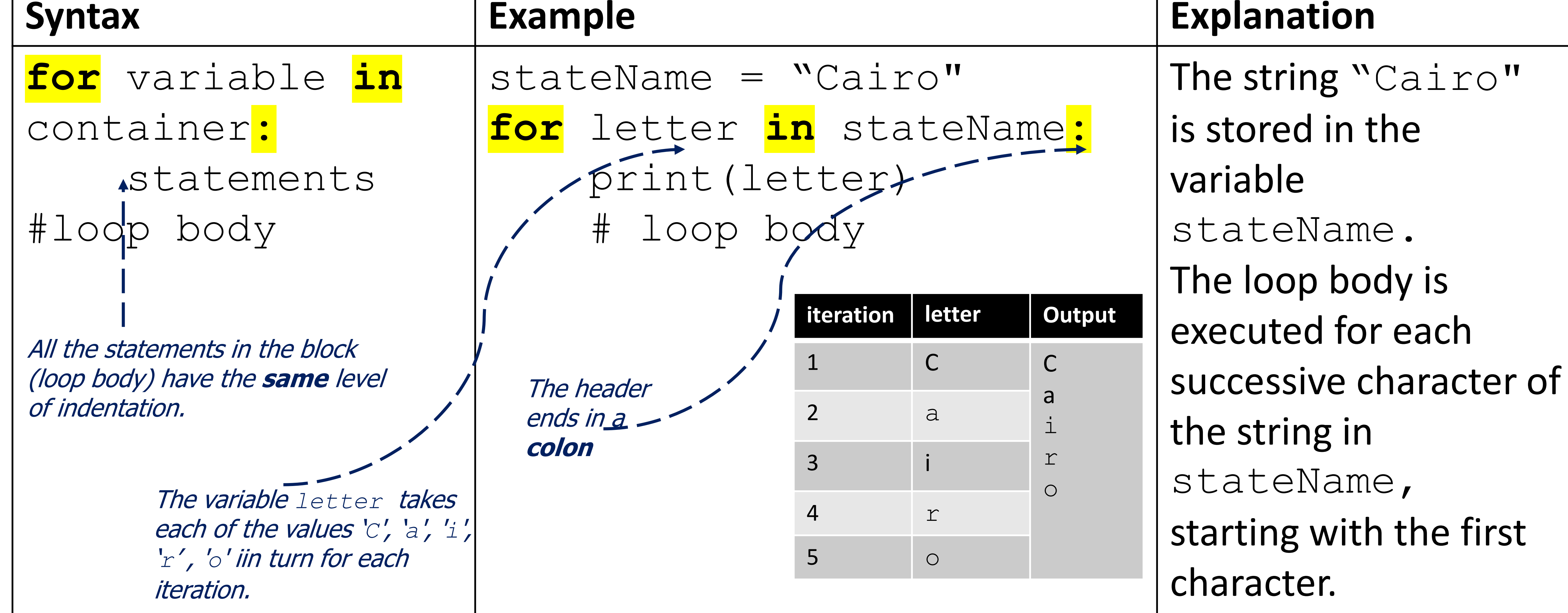
```
*
**
***
****
*****
```

```
1
12
123
1234
12345
```

```
1
11
111
1111
11111
```

Objectives of the exercises set (1)

- **Objectives**
  - Use `for` statements to implement count-controlled loops that iterate over a range of integer values or the contents of any container.



Syntax	Example	Explanation

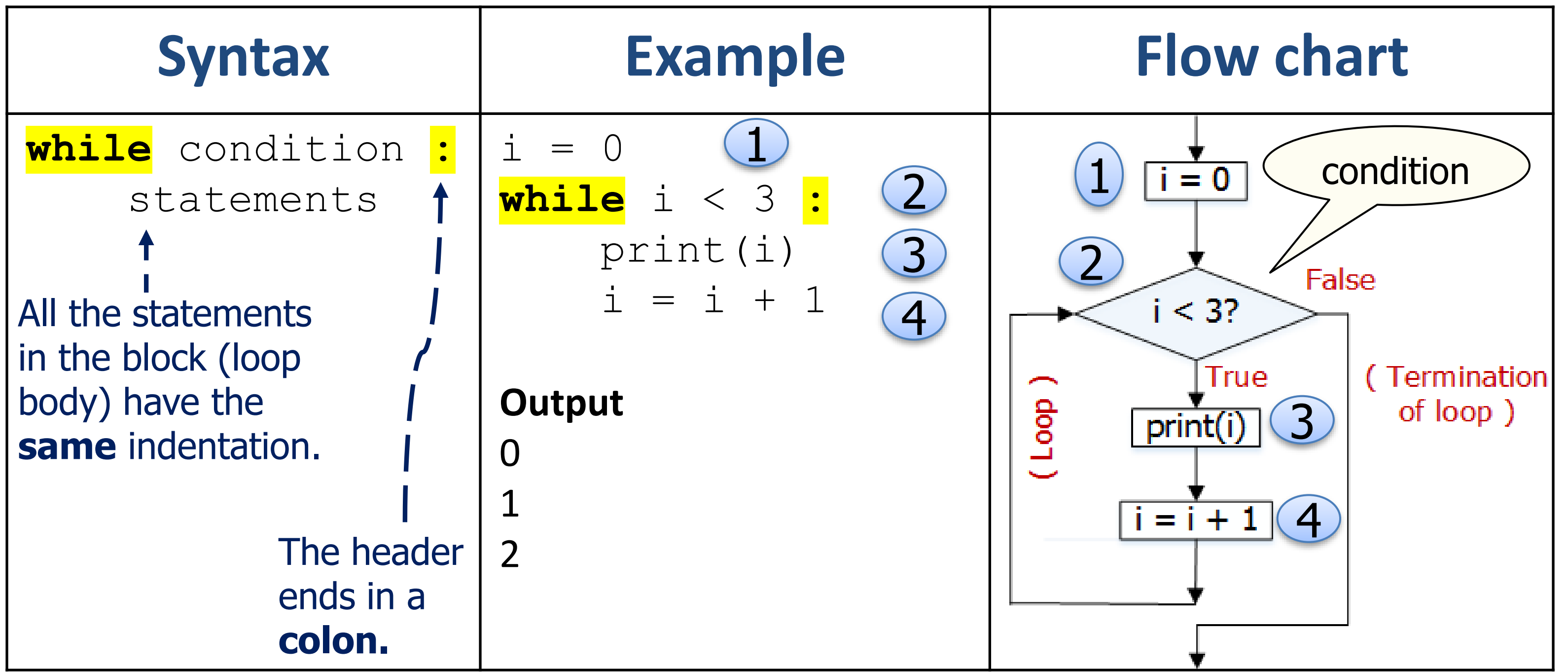
`for variable in container:`
`    statements #loop body`
	`stateName = "Cairo"`
	`for letter in stateName:`
	`    print(letter)`
	`    # loop body`
		The string `"Cairo"` is stored in the variable `stateName`. The loop body is executed for each successive character of the string in `stateName`, starting with the first character.

iteration	letter	Output
1	C	C a i r o
2	a	
3	i	
4	r	
5	o	

Objectives of the exercises set (2)

- **Objectives**
  - Use `while` statements to implement event-controlled loops.

    A `while` loop executes instructions repeatedly while a condition is true.



In the example, the variable `i` is initialised outside the while loop (statement 1) and updated in the loop body (statement 4).

Objectives of the previous exercises

– The table below shows the working of the previous `while` loop example:

i	i < 3 ?	Output using print(i)	i = i + 1
0	True	0	1
1	True	1	2
2	True	2	3
3	False – end of the `while` loop		

The break Statement

- Causes an exit from anywhere in the body of a loop.
- When **`break`** is executed.
  - Loop immediately terminates.
- Break statements usually occur in *if* statements.

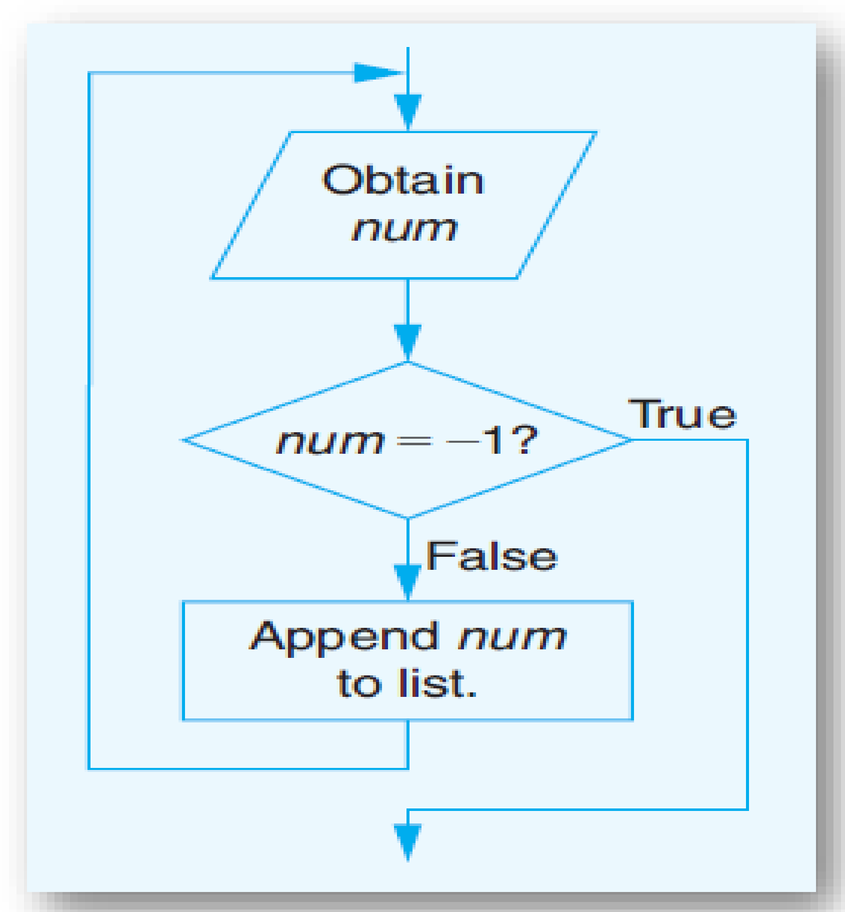
The break Statement

- Example 6: Program uses *break* to avoid two input statements.

```
## Obtain list of numbers.
list1 = []
while True:
    num = eval(input("Enter a nonnegative number: "))
    if num == -1:
        break    # Immediately terminate the loop.
    list1.append(num)
```

The break Statement

Flowchart for previous Example



The continue Statement

- When **`continue`** executed in a while loop:
  - Current iteration of the loop terminates .
  - Execution returns to the loop’s header.
- Usually appear inside *if* statements.

Infinite Loops

- Condition ***number >= 0*** always true.

```
## Infinite loop.
print("(Enter -1 to terminate entering numbers.)")
number = 0
while number >= 0:
    number = eval(input("Enter a number to square: "))
    number = number * number
    print(number)
```

Control Structures

if *condition*:
    statements
[elif *condition*:
    statements] ...
else:
    statements

while *condition*:
    statements

for *var* in *sequence*:
    statements

break
continue

Looping Through a Set

```
print('Before')
for thing in [9, 41, 12, 3, 74, 15] :
    print(thing)
print('After')
```

```
$ python basicloop.py
Before
9
41
12
3
74
15
After
```

What is the Largest Number?

What is the Largest Number?

3

What is the Largest Number?

41

What is the Largest Number?

12

What is the Largest Number?

9

What is the Largest Number?

74

What is the Largest Number?

15

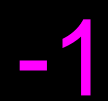
What is the Largest Number?

What is the Largest Number?

3 41 12 9 74 15

What is the Largest Number?

largest_so_far



What is the Largest Number?

3

largest_so_far

3

What is the Largest Number?

41

largest_so_far | 41

What is the Largest Number?

12

largest_so_far | 41

What is the Largest Number?

9

largest_so_far 41

What is the Largest Number?

74

largest_so_far

74

What is the Largest Number?

15

74

What is the Largest Number?

3 41 12 9 74 15

74

Finding the Largest Value

```
largest_so_far = -1
print('Before', largest_so_far)
for the_num in [9, 41, 12, 3, 74, 15] :
   if the_num > largest_so_far :
      largest_so_far = the_num
   print(largest_so_far, the_num)

print('After', largest_so_far)
```

```
$ python largest.py
Before -1
9 9
41 41
41 12
41 3
74 74
74 15
After 74
```

We make a variable that contains the largest value we have seen so far. If the current number we are looking at is larger, it is the new largest value we have seen so far.