DFN40323 PROGRAMMING ESSENTIALS IN PYTHON

Lecturer:
Pn. Sharizan Binti Abdul Jamil

CHAPTER 2:
MAKING DECISIONS IN PYTHON

Lesson Learning Outcome:

1

Explain relational operator

a. Relational operator
b. Logical operator
c. Bitwise operator

2

Explain conditional operator

a. Conditions statements
b. Looping statements

3

Manipulate Lists for Python

a. List methods
b. List elements

4

Construct List in simple program

Python Relational Operators/ Comparison Operators

Operator	Description
==	If values of two operands are equal, then the condition becomes true.
!=	If values of two operands are not equal, then condition becomes true.
<>	If values of two operands are not equal, then condition becomes true.
>	If value of left operand is greater than the value of right operand, then condition becomes true.
<	If value of left operand is less than the value of right operand, then condition becomes true.

Python Relational Operators/ Comparison Operators (cont.)

Operator	Description
>=	If value of left operand is greater than or equal to the value of right operand, then condition becomes true.
<=	If value of left operand is less than or equal to the value of right operand, then condition becomes true.

Python Logical Operators

- Logical operators are used to combine conditional statements:

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

Python Bitwise Operators

- Bitwise operator works on bits and performs bit-by-bit operation.
- Assume if

  a = 60

  b = 13
- Now in binary format, they will be as follows:

  a = 0011 1100

  b = 0000 1101
- Python's built-in function **bin()** can be used to obtain binary representation of an integer number.

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

Python Bitwise Operators (cont.)

- Let's refresh want have you learned before:

X	Y	X & Y	X | Y	X ^ Y
0	0			
0	1			
1	0			
1	1			

Answer:

LAB ACTIVITY 3 (i)

CONDITION STATEMENTS

Python Conditions and IF Statements

- Python supports the usual logical conditions from mathematics:
  - Equals: a == b
  - Not Equals: a != b
  - Less than: a < b
  - Less than or equal to: a <= b
  - Greater than: a > b
  - Greater than or equal to: a >= b
- These conditions can be used in several ways, most commonly in “if statements” and loops.
- An “ if statement” is written by using the **if** keyword.

“if” STATEMENT

- If statement consists of a Boolean expression followed by one or more statements.
- Syntax:

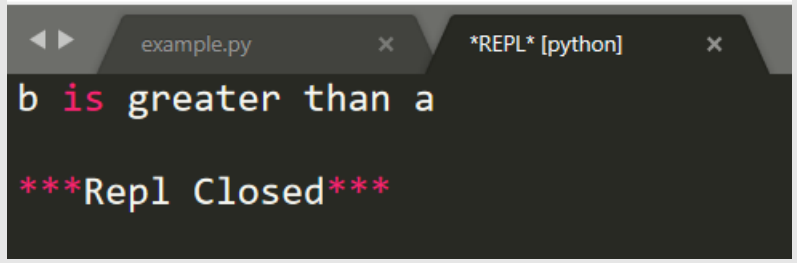
  if expression:

  statement(s)
- Example:

```
a = 33
b = 200
if b > a:
print (“b is greater than a”)
```

Python relies on indentation (whitespace at the beginning of a line) to define scope in the code. Other programming languages often use curly-brackets for this purpose.

Output:



“else” STATEMENT

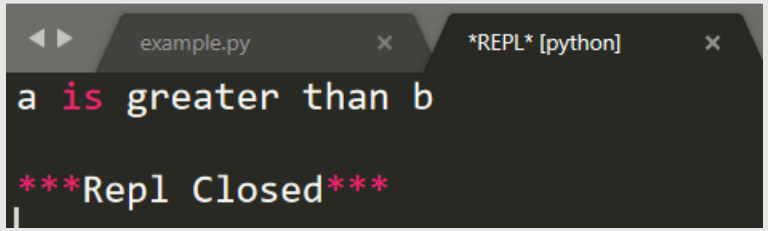
- The else keyword catches anything which isn’t caught by the preceding conditions.
- Syntax:

```
if expression:
 statement(s)
else:
 statement(s)
```

- Example:

```
a = 200
b = 33
if b > a:
 print (“b is greater than a”)
else:
 print (“a is greater than b”)
```

Output:



“elif” STATEMENT

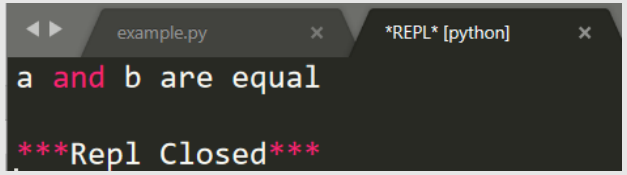
- The elif keyword is pythons way of saying “if the previous conditions were not true, then try this condition”.
- Syntax:

```
if expression1:
 statement(s)
elif expression2:
 statement(s)
else:
 statement(s)
```

Example:

```
a = 33
b = 33
if b > a:
 print (“b is greater than a”)
elif a == b:
 print (“a and b are equal”)
else:
 print (“a is less than b”)
```

Output:



LOOPING STATEMENTS

Python Looping Statements

- Python has two primitive loop commands:
  - while loops
  - for loops

The 'while' Loop

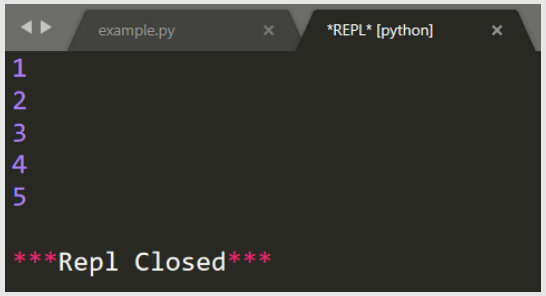
- With the while loop, we can execute a set of statements as long as a condition is true or
- Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.
- Syntax:

```
while expression:
 statement(s)
```

Example:

```
i = 1
while i < 6:
 print(i)
i += 1
```

Output:



Remember to increment i, or else the loop will continue forever.

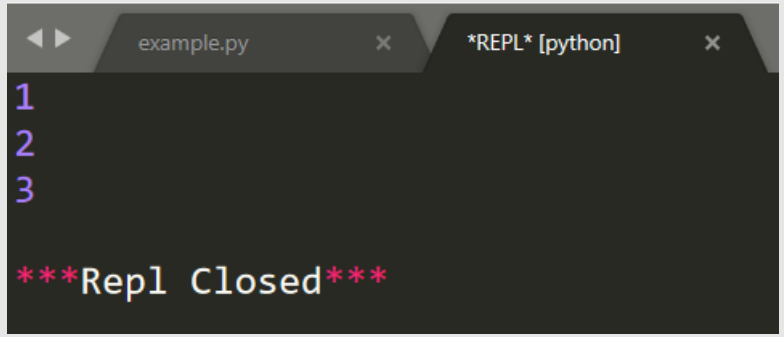
The 'break' Statement

- With the break statement, we can stop the loop even if the while condition is true or
- Terminates the loop statement and transfers execution to the statement immediately following the loop.
- Example:

```
i = 1
while i < 6:
 print(i)
 if i == 3:
  break
 i += 1
```

Exit the loop when i is 3

Output:



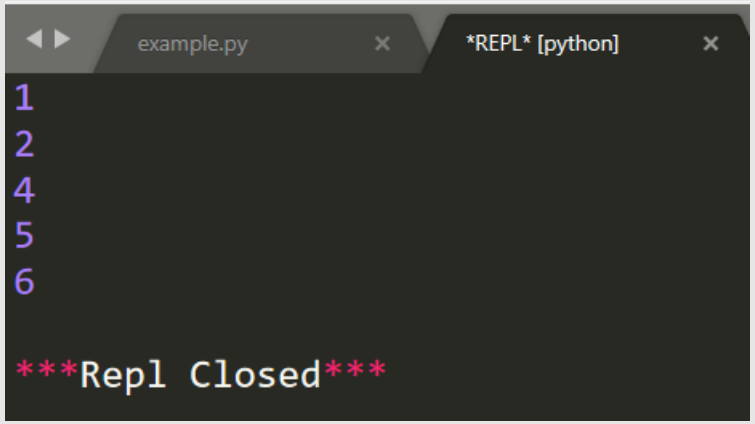
The 'continue' Statement

- With the continue statement, we can stop the current iteration and continue with the next or
- Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
- Example:

```
i = 0
while i < 6:
 i += 1
 if i == 3:
  continue
 print (i)
```

Continue to the next iteration if i is 3

Output:



The 'else' Statement

- With the else statement, we can run a block of code once when the condition no longer is true
- Example:

```
i = 1
while i < 6:
 print (i)
 i += 1
else:
 print ("i is no longer less than 6")
```

Print a message once the condition is false

Output:

Python ‘for’ Loops

- A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).
- This is less like the for keyword in other programming languages, and works more like an iterator method as found in other object-oriented programming languages.
- With the for loop we can execute a set of statements, once for each item in a list, tuple, set etc.
- Use keywords **“break”** to quit the for loop
- Use keywords **“continue”** to skip current loop and continue next loop
- Use **range function** to set loop start, loop end and loop step

  range (start, end, step/increment/decrement)

Python ‘for’ Loops (cont.)

- Syntax:

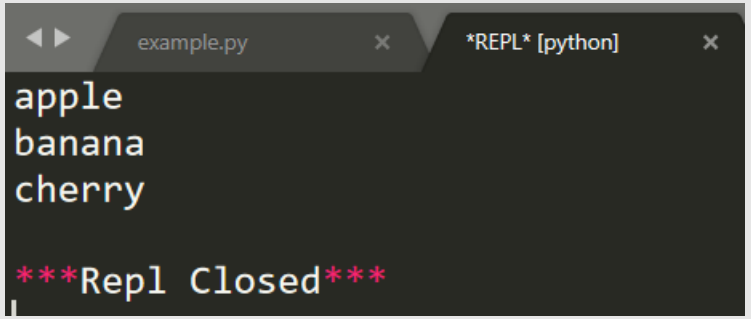
```
for variable in some-sequence-type:
 python-statements
else:
 python-statements
```

- Example: Print each fruit in a fruit list

```
fruits = [“apple”, “banana”, “cherry”]
for x in fruits:
 print (x)
```

The for loop **does not require and indexing variable** to set beforehand.

Output:



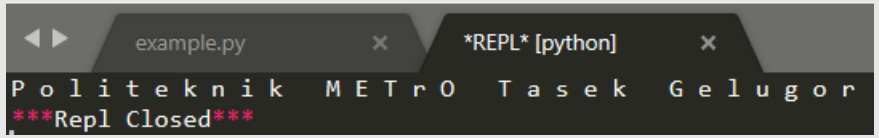
Looping Through a String

- Even strings are iterable objects, they contain a sequence of characters:

```
for x in "Politeknik METrO Tasek Gelugor":
 print (x)
```

- Output:

How to get this?



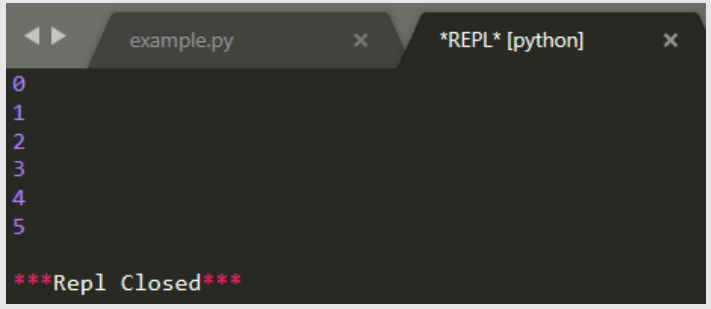
The range() Function

- The loop through a set of code a specified number of times, we can use the range() function,
- The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.
- Example:

```
for x in range(6):
 print (x)
```

Note that range(6) is not the values of 0 to 6, but the **values 0 to 5**.

- Output:

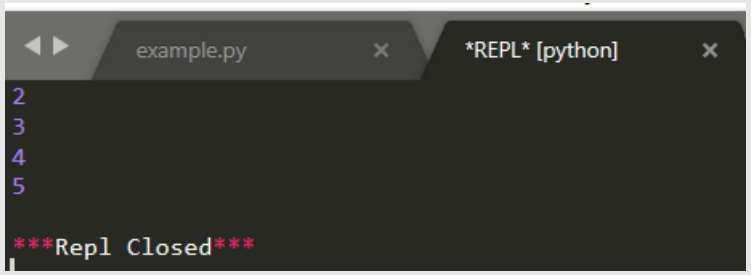


The range() Function (cont.)

- The range() function defaults to 0 as a starting value, however it is possible to specify the starting value by adding a parameter.
- Example: range(2,6), which means the values from 2 to 6 (but **NOT** including **6**)

```
for x in range(2,6):
 print (x)
```

- Output:



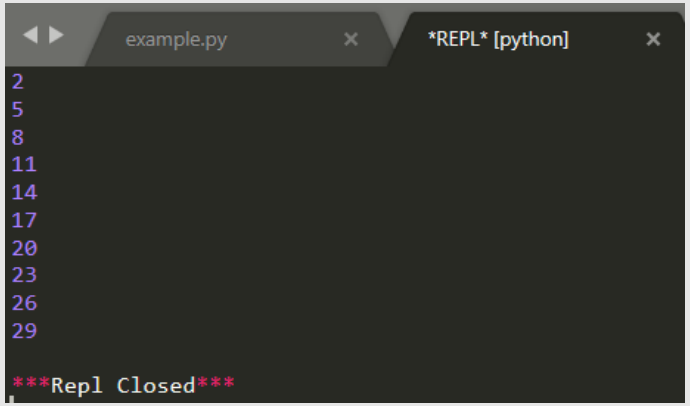
The range() Function (cont.)

- The range() function defaults to increment the sequence by 1, however it is possible to specify the increment value by adding a third parameter.
- Example: range(2,30,3)

```
for x in range(2,30, 3):
 print (x)
```

Increment the sequence with 3 (default is 1):

- Output:



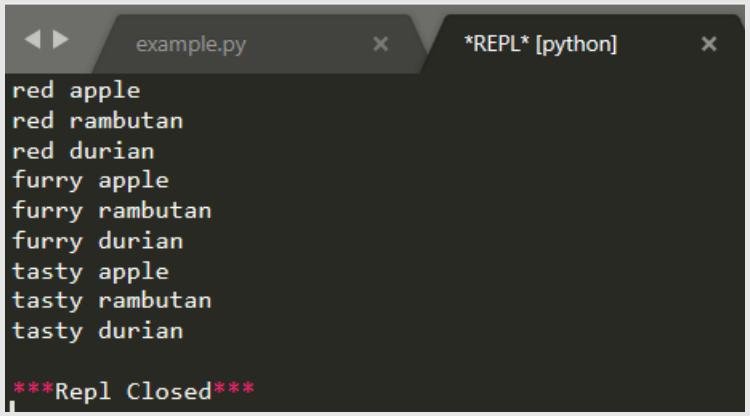
Nested Loops

- A nested loop is a loop inside a loop.
- The "**inner loop**" will be executed one time for each iteration of the "**outer loop**".
- Example:

```
adj = ["red", "furry", "tasty"]
fruits = ["apple", "rambutan", "durian"]

for x in adj:
        for y in fruits:
                print (x, y)
```

Output:



LAB ACTIVITY 3 (ii)