AND என் கிறவர்ரரொம்ப ரெகட்டிவ் ரபர்சன். குற்றம் கொண் பவர். Falseயை மட்டுமம மேடுபவர். He searches for the first false.

ஒமர ஒரு False பொ◌ேர்ொலும் ரெேமா்மொக falseனுரசொல்லிட்டு மபொயிருவொர்.

ஒரு true கியேெட்ால் மசொகமொகிவிடுவொர். அடேு்து false கியடக்குமொ என்று பொர்ப்பொர். கயடசி வயர பொ◌ேர்தும் true◌ோன் என்

றொல், மவண் டொ ரவறுப்பொக ெேஅ்
கயடசி trueயவ ரசொல்லுவொர்.
Last true.

OR ரரொம்ப ெ◌ல்லவர்
எப்மபொதும் trueயவ மேடுவொர். He searches for true. ஒரு true கியேேெட்ாலும்அவருக்கு மபொதும். First true.
ஒரு மவயை false ெேெவ்ால்கூட மீண்டும் ெைவார்ப்பு ரகொேடுேது trueயவ

மேடுவொர்.
எல்லொமம false ஆகி மபொனொல்
வழிஇல்லொமல் கயடசி falseயை
ரசொல்லுவொர். Last false.

AND - bad guy - First false or Last true.

OR - good guy - First true or Last false.

- From Data Science – Muthuvel
- --------------

Section 14.2: if, elif, and else

In Python you can define a series of conditionals using if for the first one, elif for the rest, up until the final (optional) else for **anything not caught by the other conditionals.**

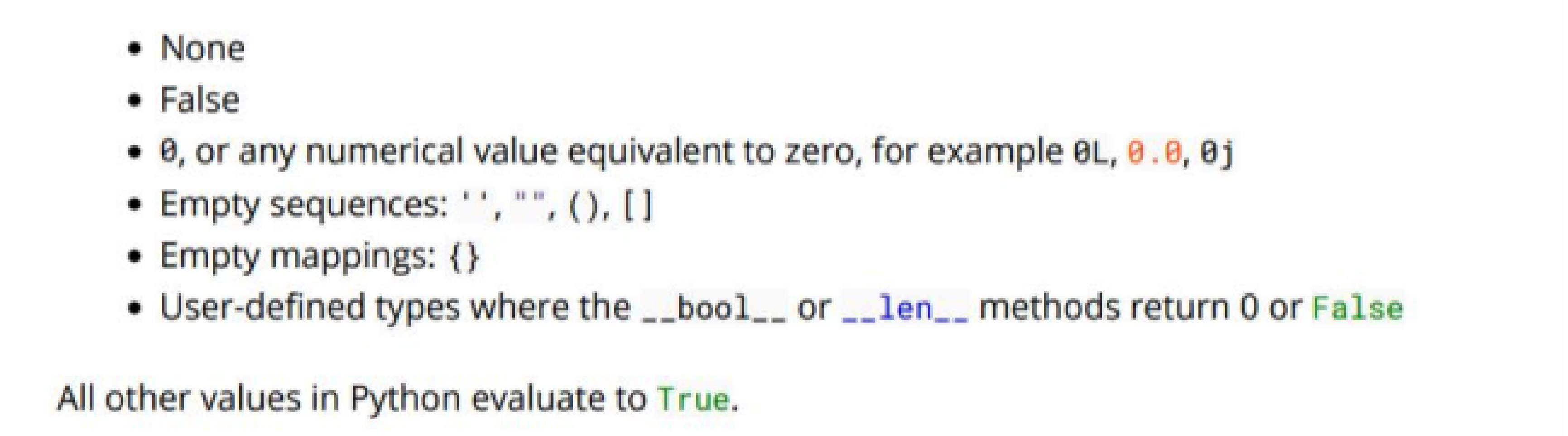
```
number = 5

if number > 2:
 print("Number is bigger than 2")
elif number < 2:
 print("number is smaller than 2")
else:
 print("Number is 2")

output
Number is bigger than 2
==========
```

Using else if instead of elif will trigger a syntax error and is not allowed.

==========

Section 14.3: Truth Values

The following values are considered **falsey**, in that they evaluate to False when applied to a boolean operator.



- None
- False
- 0, or any numerical value equivalent to zero, for example 0L, 0.0, 0j
- Empty sequences: '', "", (), []
- Empty mappings: {}
- User-defined types where the __bool__ or __len__ methods return 0 or False

All other values in Python evaluate to True.

```
print(int.__bool__(10))  #True
print("dd".__len__())  #2
print(int.__bool__(0))#False
print("".__len__())#0
==========
if True:
 print("Yes")
if False:
 print("it does not print and execution does not come to
this line")
```

```
output
Yes
=======
if None:
 print("it does not print and execution does not come to this
line")
=========
if 1:
 print("Yes")
if 0:
 print("it does not print and execution does not come to this
line")
output
Yes
===========
if [3,4,5]:
 print("Yes")
if []:
 print("it does not print and execution does not come to this
line")
---------
if (5,6):
 print("Yes")
if ():
 print("it does not print and execution does not come to this
line")
----------

if {7,8,}:
 print("Yes")
if {}:
 print("it does not print and execution does not come to this
line")
========
if 'abc':
 print("Yes")
if '':
 print("it does not print and execution does not come to
this line")
-------

if "Data Science":
 print("Yes")
```

```
if "":
 print("it does not print and execution does not come to this line")
----------
if 5-5:
 print("it does not print and execution does not come to this line")

===========
if 0.0:
 print("it does not print and execution does not come to this line")

if 1j:
 print("Yes")

if 0j:
 print("it does not print and execution does not come to this line")

if 0L:
 print("it does not print and execution does not come to this line")#Note: it gives syntax error. Doubt
===============
```

Note: A common mistake is to simply check for the Falseness of an operation which returns different Falsey values where the difference matters. For example, using if foo() rather than the more explicit if foo() is None

```
def foo():
 return " some string"

if foo():
 print("print somthing")
```

output

print something

```
=======
def foo():
 return None

if foo():
 print("this line does not execute")#None becomes Falsey
-----------
def foo():
 return True
```

```
if foo():
 print("this line will execute")
------------
def foo():
 return False
if foo():
 print("this line does not execute")
===========
```

Section 14.4: Boolean Logic Expressions

Boolean logic expressions, in addition to evaluating to True or False, return the value that was interpreted as True or False. It is Pythonic way to represent logic that might otherwise require an if-else test.

```
def foo():
 return None

if foo():#they calling the foo()in if statement
 print("this line does not execute")
----------
```

```
def foo():
 return True

if foo():
 print("this line will execute")
------------
def foo():
 return False
if foo():
 print("this line does not execute")
```

And operator

The and operator evaluates all expressions and **returns the last expression** if all expressions evaluate to True. Otherwise it returns the first value that evaluates to False:

```
print(1 and 2 and 3 and 4)  # 4
print("P" and "C" and "M" and "PASS")
```

- Mandatroy fields are in ‘and’ ..

Ex (while filling the application)

```
print(1 and 2 and 3)  # 3
print(1 and 2 and 0 and 4) # 0
print(1 and 2 and [] and 4) # []
print(1 and {} and [] and 4) #{}// it returns the first
occurance of the Falsey
=======
print(2 and "Data Science")#Data Science
print(2 and "Data Science" and "AI") # AI
print("" and "Py")#"" // we will not be able to view “”
-----
a = "" and "Py"
print(repr(a))# “”
=========
```

Or operator

The or operator evaluates the expressions left to right and returns the first value that evaluates to True or the last value (if none are True).

```
print(1 or 2)#1
print(0 or 2) #2
print(None or 5)#5
print(False or 6 )#6
print(True or False)#True
print(False or True)#True
print([] or True)#True
print([] or 0 or {})#{} # takes the last False
print([] or "ABC" or {22})#ABC (Takes first True value)
print(None or None)# it takes the last None
print(0 or {} or [])#[] ..it takes the last false

print(0 or (10-10) or "BB")  #BB
```

Lazy evaluation

When you use this approach, remember that the **evaluation is lazy. Expressions that are not required to be evaluated to determine the result are not evaluated**. For example:

```
def print_me():
 print("I am here")
```

```
0 and print_me()
output
```

Note: it does not call the print_me(). Because the expression starts with 0(zero) that stands for False

What circumstances can we use?

In the above example, **print_me** is never executed because Python can determine the entire expression is False when it encounters the 0 (False). Keep this in mind if print_me needs to execute to serve your program logic.

If we use or in the expression then the print_me() will execute – see below

```
def test1():
 # print("I am here")
 return 'ABC'

def test2():
 # print('True')
 return [5, 6]

print(test1() or test2())
print('ABC' or [])
```

o/p:

ABC

ABC

```
def print_me():
 print("I am here")

0 or print_me()
output
I am here
```

```
def print_me():
 # print("I am here")
 return 'Abc'

print(1 and 2 and print_me() and 5)
print(1 and 2 and 'Abc' and 5)
```

o/p:

5

5

=======

See the below code for more info

```
def print_me():
 print("I am here")

print(1 and 2 and print_me() and 5)
```

```
output
I am here
None
```

Note: technically, we should get 5 as output(becose 'and' checks if all conditions are true, then it gives as the last value(here the last value is 5). But in betweeen we call function(print_me()). Print_me() returns None (Which is false and **first false**) So prints the "I am here" then prints None

------

Another example to NOT call the fn. we converted the print_me() as string. Now the last value 5 will be printed

```
def print_me():
 print("I am here")

print(1 and 2 and "print_me()" and 5)
```

```
>>5
```

========

See some more ex(Mel)

```
def print_me():
 print("I am here")
 return False

print(1 and 2 and print_me() and 10)
```

output

```
I am here
False
```

Note: the fn is called / result is printed. And the return value also printed. Since the return value is false, it gives first False value

----

```
def print_me():
 print("I am here")
 return True

print(1 and 2 and print_me() and 10)
```

output

```
I am here
```

```
10
```

Note: the fn is called and printed the result. Since it is True, it goes and check the next value (ie 10). Now all values become True.Therefore, it gives the last value in the expression .

How None works

```
def print_me():
 print("I am here")

print(1 and 2 and print_me() and 10)
output
I am here
None
```

Note: here the fn is called and printed. This fn does not return anything. So by default the return type is None. The None is retured to the expreission. Since is None stands for False, the expression shows the false value, here the false value is None. So we get None

===========

Testing for multiple conditions

A common mistake when checking for multiple conditions is to apply the logic incorrectly.

This example is trying to check if two variables are each greater than 2. The statement is evaluated as - if (a) and (b > 2). This produces an unexpected result because bool(a) evaluates as True when a is not zero.

```
a = 1
b = 6

if a and b > 2:
 print("Yes")
else:
 print("No")
```

output

```
Yes
```

Note: but this output is not right.

=============

Each variable needs to be compared separately. See below

```
a = 1
b = 6

if a > 2 and b > 2:
 print("Yes")
else:
 print("No")
```

output

No #this is the right output

=========

Another, similar, mistake is made when checking if a variable is one of multiple values. The statement in this example is evaluated as - if (a == 3) or (4) or (6). This produces an unexpected result because bool(4) and bool(6) each evaluate to True

```
a = 1
if a ==3 or 4 or 6:
 print("Yes")
else:
 print("No")
Output
Yes
```

Note: but this output is not right.

========

Again each comparison must be made separately

```
a = 1
if a == 3 or a == 4 or a == 6:
 print("Yes")
else:
 print("No")
output
No
```

=======

Using the in operator is the canonical way to write this. (using membership operator)

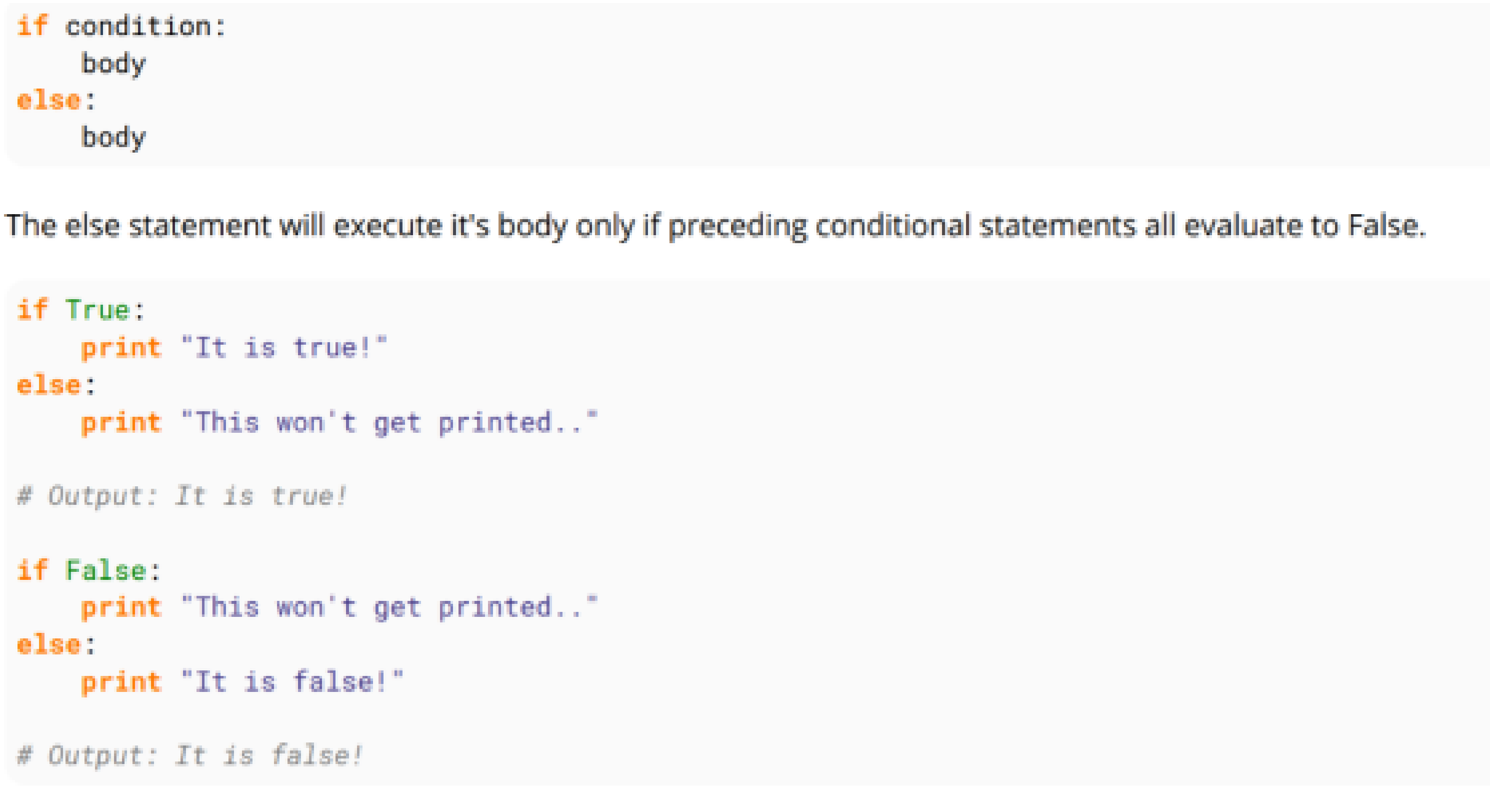
```
a = 1
if a in (3,4,6):
 print("Yes")
else:
 print("No")
>>No
===============
==============
```

Section 14.6: Else statement

```
if True:
 print("it prints")
else:
 print("it does not print")

if False:
 print("it prints")
else:
 print("it does not print")
```

==========

Note: Try with ‘not’ (not True / not False). We get the opposite result of the above



```
if condition:
    body
else:
    body
```

The else statement will execute it's body only if preceding conditional statements all evaluate to False.

```
if True:
    print "It is true!"
else:
    print "This won't get printed.."

# Output: It is true!

if False:
    print "This won't get printed.."
else:
    print "It is false!"

# Output: It is false!
```

Section 14.7: Testing if an object is None and assigning it

You'll often want to assign something to an object if it is None, indicating it has not been assigned. We'll use aDate.

The simplest way to do this is to use the **is** None test.

```
import datetime
aDate = None
if aDate is None:
 aDate =datetime.datetime.now()
print(aDate)
```

(Note that it is more Pythonic to say is None instead of == None.)

But this can be optimized slightly by exploiting the notion that not None will evaluate to True in a boolean expression. The following code is equivalent:

```
import datetime
aDate = None
if not aDate is None:
 aDate =datetime.datetime.now()
 print(aDate)
no output
```

even if we don’t have money at our bank, using overdraft facility, still we will be able to withdraw money ------

```
import datetime
aDate = None # None means False
if not aDate: # here we convert the None to True by adding ‘not’
 aDate =datetime.datetime.now()
 print(aDate)
```

output

```
2020-10-31 00:36:23.329926
```

========

But there is a more Pythonic way. The following code is also equivalent:

```
import datetime
aDate = None
aDate = aDate or datetime.date.today()
print(aDate)
```

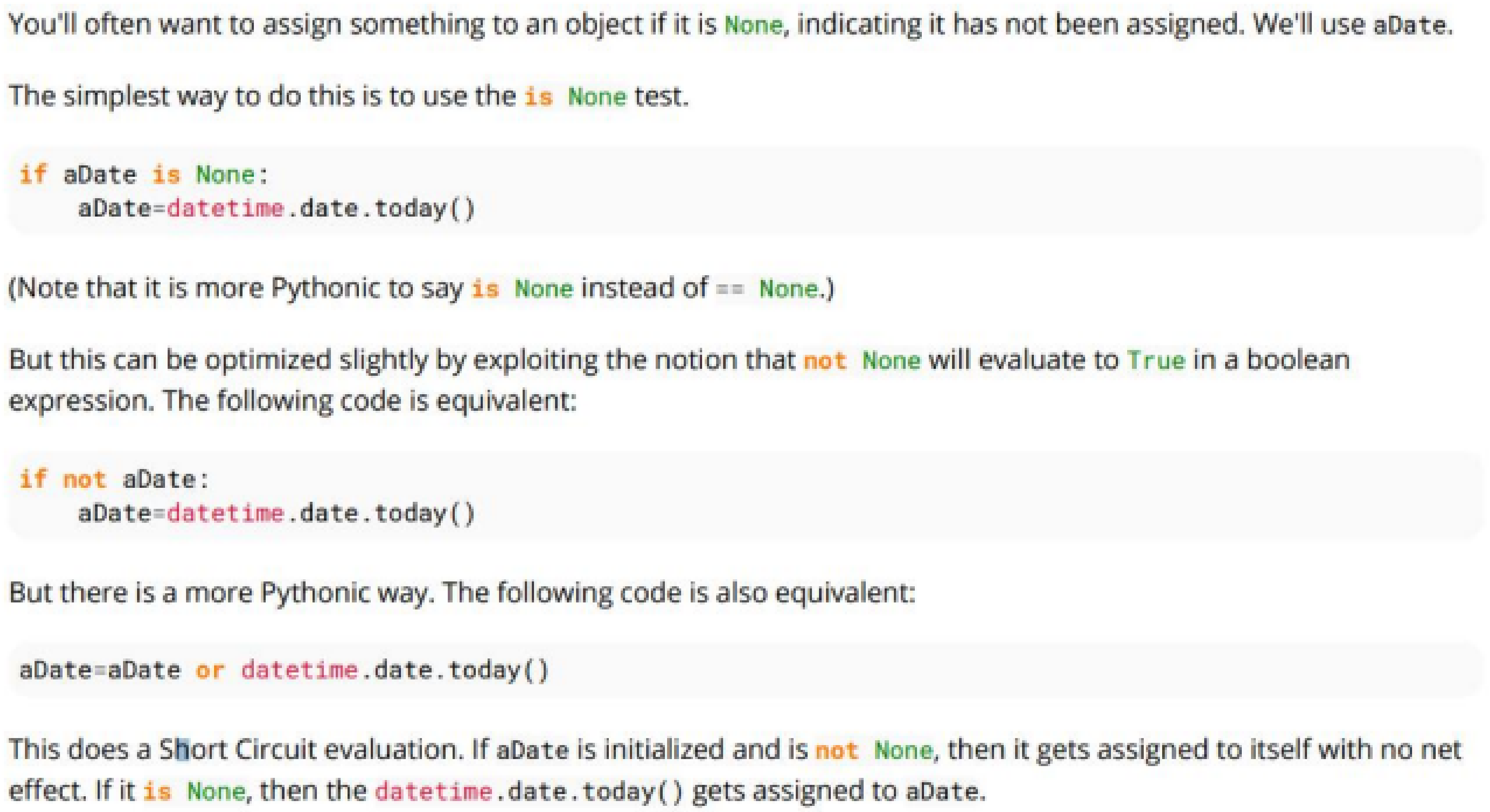
output

```
2020-10-31
```

========

(the above) This does **a Short Circuit evaluation**. If aDate is initialized and is not None, then it gets assigned to itself with no net effect. If it is None, then the datetime.date.today() gets assigned to aDate

========



You'll often want to assign something to an object if it is None, indicating it has not been assigned. We'll use aDate.

The simplest way to do this is to use the is None test.

```
if aDate is None:
    aDate=datetime.date.today()
```

(Note that it is more Pythonic to say is None instead of == None.)

But this can be optimized slightly by exploiting the notion that not None will evaluate to True in a boolean expression. The following code is equivalent:

```
if not aDate:
    aDate=datetime.date.today()
```

But there is a more Pythonic way. The following code is also equivalent:

```
aDate=aDate or datetime.date.today()
```

This does a Short Circuit evaluation. If aDate is initialized and is not None, then it gets assigned to itself with no net effect. If it is None, then the datetime.date.today() gets assigned to aDate.

Section 14.8: If statement