Forward School

Program Code: J620-002-4:2020

Program Name: FRONT-END SOFTWARE DEVELOPMENT

Title : Logic and Repetition

Name: Ooi Caaron

IC Number: 990701-07-5837

Date : 24/6/23

Introduction : Learning the logic and repetition

Conclusion : Still need to do more practice

Type *Markdown* and LaTeX: α^2

Module P02: Logic and Repetition

In this module, we'll first understand what procedures are and why you might want to use them.
Then you'll start to write some procedures. Next, we'll learn about Python logic, or how to make comparisons.

Finally, we'll go on to repetition, and how to repeat operations in Python using while and for loops.

Procedural Abstraction

What is a procedure? Why might you want to abstract it?

We use procedures because we want to stop us from doing tedious work.
Anything we might want to do again and again we will want to 'abstract'.

Functions

A function is a block of organized, reusable code that is used to perform a single, related action. For example, we can use the same procedure to multiply two numbers, whatever they are. We then specify the numbers being multiplied as arguements of the function. Many functions return a value which you can then use; they can also print something out immediately

Syntax:

In [1]:

```
def functionname( parameters ):
    "functionname ... syntax: blabla"
    block_of_code
    return [expression]
```

Notice that you can add some help text after the function header line, so that you can print the help text by:

In [2]:

```
print((functionname.__doc__))
```

```
functionname ... syntax: blabla
```

In [2]:

```
# void: does not return
def myPrinter(text):
    print(text)

myPrinter(234)

# has return
def multiplyer(i, j):
    return i*j

print(multiplyer('hello', 3))

print(multiplyer(4, 3) + 1)

multiplyer(3, 4) # return nothing
```

```
234
hellohellohello
13
```

Out[2]:

```
12
```

In [6]:

```
def print_info( name, age = 35, height=150, weight=60 ):
    "This prints a passed info into this function"
    print("Name: ", name)
    print("Age ", age)
    print("Height ", height)
    print("Weight ", weight)
    print()

print_info('he', 22, weight=65)

print_info('she')
```

```
Name:  he
Age  22
Height  150
Weight  65

Name:  she
Age  35
Height  150
Weight  60
```

Quick Exercise 1 Let's revisit the Farenheit to Celsius conversion task. Write a function `'F2C'` that converts a Fahrenheit value to Celsius, and returns the value.

In [13]:

```
def ftoc(f):
    print((f - 32) * (5/9))

ftoc(100)
```

```
37.77777777777778
```

Logic

Can a computer think? How does a computer think differently from how we do? We've seen how computers can represent abstract concepts. One of those abstract concepts is logic. We've developed programs that can use logic.

A **boolean** expression is an expression that is either *true* or *false*. The following examples use the operator ==, which compares two operands and produces True if they are equal and False otherwise.

There are three logical operators: *and*, *or*, and *not*. The semantics (meaning) of these operators is similar to their meaning in English.

String logic

is -- object identity

is not -- negated object identity

x in s -- True if an item of s is equal to x, else False

x not in s -- False if an item of s is equal to x, else True

x or y -- if x is false, then y, else x

x and y -- if x is false, then x, else y

not x -- if x is false, then True, else False

Arithmetic logic

> -- Strictly larger than

== -- Is the identity of

>= -- Greater than or equal to

!= -- Is not the identity of.

In [14]:

```
print((5 > 2))
print((2 > 5))
print((2 is 2))
print((2 == 3))
print((5 > 2 or 2 > 1))
print((5 > 2 and 2 > 2))
print(('s' in 'datascience'))
print(('x' in 'datascience'))
```

```
True
False
True
False
True
False
True
False

<>:3: SyntaxWarning: "is" with a literal. Did you mean "=="?
<>:3: SyntaxWarning: "is" with a literal. Did you mean "=="?
<ipython-input-14-56adce85b2e3>:3: SyntaxWarning: "is" with a literal. Did
you mean "=="?
  print((2 is 2))
```

In [15]:

```
# Why is == equals, not = ? Test it out
print((2 = 3))
```

```
  File "<ipython-input-15-43fe0a13c0d0>", line 2
    print((2 = 3))
             ^
SyntaxError: invalid syntax
```

Conditional Execution

Python does not use { } to enclose blocks of code for if/loops/function etc. like C. Instead, Python uses the colon (:) and indentation/whitespace to group statements.

In order to write useful programs, we almost always need the ability to check conditions and change the behavior of the program accordingly. Conditional statements give us this ability. The simplest form is the if statement:

```
if x > 0 :
    print 'x is positive'
```

The boolean expression after the if statement is called the condition. We end the if statement with a colon character (:) and the line(s) after the if statement are indented.

If the logical condition is true, then the indented statement gets executed. If the logical condition is false, the indented statement is skipped.

Alternative execution

A second form of the if statement is alternative execution, in which there are two possibilities and the condition determines which one gets executed. The syntax looks like this:

```
if x%2 == 0 :
    print 'x is even'
else :
    print 'x is odd'
```

If the remainder when x is divided by 2 is 0, then we know that x is even, and the program displays a message to that effect. If the condition is false, the second set of statements is executed.

Since the condition must be true or false, exactly one of the alternatives will be executed. The alternatives are called branches, because they are branches in the flow of execution.

Chained conditionals

Sometimes there are more than two possibilities and we need more than two branches. One way to express a computation like that is a chained conditional:

```
if x < y:
    print 'x is less than y'
elif x > y:
    print 'x is greater than y'
else:
    print 'x and y are equal'
```

elif is an abbreviation of "else if". Again, exactly one branch will be executed.

There is no limit on the number of elif statements. If there is an else clause, it has to be at the end, but there doesn't have to be one.

In [20]:

```
if False:
    print("block of code 1")
elif True:
    print("block of code 2")
else:
    print("block of code 3")
```

```
block of code 2
```

In [27]:

```
speed = 105
mood = 'terrible'

if speed >= 80:
    print('License and registration please')
    if mood == 'terrible' and speed >= 100:
        print('You have the right to remain silent.')
    elif mood == 'bad' or speed >= 90:
        print("I'm going to have to write you a ticket.")
    else:
        print("Let's try to keep it under 80 ok?")
```

```
License and registration please
You have the right to remain silent.
```

In [31]:

```
#string = 'Hi there' # True example
string = 'Good bye' # False example

result = string.find('th')
print(result)

if result != -1:
    print('Success!')
else:
    print('Not found!')
```

```
-1
Not found!
```

In [17]:

```
# Define starts_with_B('Boyce')

def starts_with_B(name):
    if name[0] == 'B' or name[0] == 'b':
        print("Yes")

# call the function
starts_with_B('Boyce')
```

In [51]:

```
# Define bigger(a,b)
def bigger(a, b):
    if a > b:
        return a
    else:
        return b

    # quick alternative
    #return max(a, b)

bigger(45, -64)
```

Out[51]:

```
45
```

Quick Exercise 2 Nested if problem: Define a function `biggest(a,b,c)` which takes the largest of the three inputs and returns the largest one.

In [15]:

```
# Write your code here
def biggest(a, b, c):
    return max(a,b,c)

biggest(9,9,9)
```

Out[15]:

```
9
```

Repetition

While

The while loop continues iterating until it's condition stops being true:

In [18]:

```
i = 0
while i < 10:
    i = i + 1 #i+=1
    print(i)
```

```
1
2
3
4
5
6
7
8
9
10
```

In [17]:

```
# What happens here?
# Try guessing. Run at your own risk! :)
#i = 0
#while i != 11:
#    i = i+2
#    print i
```

```
2
4
6
8
10
12
14
16
18
20
22
24
26
28
30
32
34
36
38
```

For

The for-loop iterates for all elements in a data structure (can be an array, list, dict, data structures that behave like iterators in general).

If we do not have a so-called container of elements, we can create a list of numbers on the go, and have the for loop to cycle through it... (but of course this has not much use, normally we will have a container of data)

In [94]:

```
for i in range(0, 10):
    print(i)
    #print(i, end=" ")         #### to print all in the same line use the 'end' option
```

```
0
1
2
3
4
5
6
7
8
9
```

A while-loop operates in a slightly different way. A condition is given to control the loop (to continue running or terminate).

In [20]:

```
basket = ['banana','apple','durian','orange','rambutan']
i=0
while i < 3:
    print(basket[i])
    i = i + 1    # this is the update to move the counter
```

```
banana
apple
durian
```

Let's now use a loop to count the number of items in the basket

In [21]:

```
# one way
c = 0
for fruits in basket:
    c = c + 1
print(c)

# another way
for i, fruits in enumerate(basket):
    i = i + 1

print(i)
len(basket)
```

```
5
5
```

Out[21]:

```
5
```

In [57]:

```
# write a for loop to sum all numbers in this list [3, 41, 12, 9, 74, 15]
sum = 0
for i in [3, 41, 12, 9, 74, 15]:
    sum = sum + i

print(sum)
```

```
154
```

In [23]:

```
# Define a function called factorial(n)
def factorial(n):
    f = 1
    while n > 1:    # keep multiplying while it has not reached 1
        f = f*n
        n = n-1
    return f

print(factorial(5))
```

```
120
```

Quick Exercise 3 Write a loop that prints out all the even numbers between 0 and 99.

In [47]:

```
# Write your code here
def even(n):
    i = 0
    for i in range(n):
        i = i + 1
        print(i%2)

print(even(99))

# for i in range(0,99, 2):
#     print(i)
```

```
1
0
1
0
1
0
1
0
1
0
1
0
1
0
1
0
1
0
1
```

In [ ]:

In [ ]:

```
```