PYTHON FOR DATA SCIENCE PROGRAMMING FUNDAMENTALS(A)

Ana-Maria Dobre

based on EDX Course

SEP 2023

1. CONDITIONS

COMPARISON OPERATIONS compare some value or operand and based on a condition, produce a Boolean.

When comparing two values you can use these OPERATORS:
==, !=, >, <, >=, <=

Examples: compare strings using the equality sign, greater than sign, inequality sign: (2)

if statement example: (3)
Notes:
- Try uncommenting the age variable
- **if age > 18** #expression that can be true or false
- **within an INDENT**, we have the expression that is run if the condition is true: **print("you can enter")**
- The statements after the if statement will run regardless if the condition is true or false

else statement example: (4)

elif statement example: (5)
Notes:
- The elif statement, short for else if, allows us to check additional conditions (if the condition statements before it are False).
- If the condition for the elif statement is True, the alternate expressions will be run.

Note: The **brackets are unnecessary** (we have a colon at the end of the line): (6)

Sometimes you want to check more than one condition at once. For example, you might want to check if one condition and another condition are both True. **Logical operators** allow you to combine or modify conditions:
and, or, not (7)

Notes:
- The and statement is only True when both conditions are true.
- The or statement is True if one condition, or both are True.
- The not statement outputs the opposite truth value

```
a = 5
(1)  a == 6

(2) a == 6
"William Wallace" == "Mel Gibson"
a > 5
a != 6

(3)
age = 19
#age = 18
if age > 18:
   print("you can enter" ) # (indent) runs if cond true
print("move on") # runs always

(4)
age = 18
# age = 19
if age > 18:
   print("you can enter" )
else:
   print("go see Meat Loaf" )
print("move on")

(5)
age = 18
if age > 18:
   print("you can enter" )
elif age == 18:
   print("go see Pink Floyd")
else:
   print("go see Meat Loaf" )
print("move on")

(6)
album_year = 1983
album_year = 1970
if album_year > 1980: // brackets unnecessary
   print("Album year is greater than 1980")
print('do something...')

(7)
if(album_year > 1979) and (album_year < 1990):
   print ("Album year was in between 1980 and 1989")
```

2. LOOPS

Generate an object that contains elements ordered from 0 to 2: (8)
Notes:
- It is helpful to think of the range object as an ordered list.
- **While in Python 2.x range returns a list, in 3.x it returns a range object**

The FOR loop enables you to execute a code block multiple times. Use FOR if you would like to print out every element in a list: (9) & (10)

FOR loop that doesn't require indexes: (11)

Use FOR loop to change the elements in list: (12)

Loop through the list and iterate on both index and element value: (13)

The WHILE loop exists as a tool for repeated execution based on a condition. The code block will keep being executed until the given logical condition returns a False boolean value: (14)

PRACTICE: We would like to iterate through list dates and stop at the year 1973, then print out the number of iterations. This can be done with the following block of code: (15)

```
(8) range(2)
// range(-5,5) if you don't want to start from 0

(9)
dates = [1982,1980,1973]
N = len(dates)
for i in range(N):
   print(dates[i])

(10) # Example of for loop
for i in range(0, 8):
   print(i)

(11)
for year in dates:
   print(year)

(12)
squares = ['red', 'yellow', 'green', 'purple', 'blue']
for i in range(0, 5):
   print("Before square ", i, 'is', squares[i])
   squares[i] = 'white'
   print("After square ", i, 'is', squares[i])

(13)
squares=['red', 'yellow', 'green', 'purple', 'blue']
for i, square in enumerate(squares):
   print(i, square)

(14)
count = 1
while count <= 5:
   print(count)
   count += 1

(15)
dates = [1982, 1980, 1973, 2000]
i = 0
year = dates[0]
while(year != 1973):
   print(year)
   i = i + 1
   year = dates[i]
print("It took ", i ,"repetitions to get out of loop.")
```

3. FUNCTIONS

FUNCTIONS can be: predefined & user-defined

The following elements define a **FUNCTION**: (16)
- block begins by **def** followed by the function name and round brackets ()
- There are input parameters / arguments that should be placed within these brackets
- You can also define parameters inside the brackets
- There is a body within every function that starts with a colon (:) and is indented
- You can also place documentation before the body
- The statement return exits a function, optionally passing back a value

We can obtain help about a function: (17)

Define function for multiplying two numbers: (18)

Notes:
- The function terminates at the return statement, while passing back a value.
- Mult(2, "Geronimo") prints Geronimo Geronimo

If there is no return statement, the function returns None: (19)
Note: Printing the function after a call reveals a None is the default return statement:

Examples of **predefined functions**: (20)

You can use a **loop** in a function: (21)

It's possible to pass default value as argument to a function.

GLOBAL VAR: a variable that is declared outside a function definition, its value is accessible and modifiable throughout the program. Note: as opposed to LOCAL VAR.

See how to create **global variables** from **within** a function: (22)

When the number of arguments are unknown for a function, they can all be packed into a tuple: (23)

Similarly, The arguments can also be packed into a dictionary: (24)

```
(16) #add 1 to a
def add(a):
   b = a + 1
   print(a, "If you add one: ", b)
   return(b)

(17) help(add)

(18)
def Mult(a, b):     # a, b are formal parameters
   c = a * b     # c is a local VAR (/bc it's def in a funct)
   return(c)
   print('This is not printed')
result = Mult(12,2)
print(result)

(19)
# Define functions, one with return value None and
other without return value
def GERO():
   print('Geronimo')
def GERO1():
   print('Geronimo')
   return(None)

Note: Both print(GERO()) and print(GERO1()) return:
Geronimo
None

(20)  album_ratings = [ 10.0, 8.5, 9.5, 7.0, 7.0 ]
print(album_ratings) as well as: sum and len

(21)
def PrintList(the_list):
   for element in the_list:
      print(element)
PrintList(['1', 1, 'Juan Alcazar', "abc"])

(22)
artist = "Juan del Diablo"
def printer(artist):
   global internal_var
   internal_var= "Andres Alcazar"
   print(artist,"is an artist")
printer(artist)
printer(internal_var)

(23)
def printAll(*args):
   print("No of arguments:", len(args))
   for argument in args:
      print(argument)
printAll('Horsefeather','Adonis','Bone')

(24)
def printDictionary(**args):
   for key in args:
      print(key + " : " + args[key])
printDictionary(Country='CA',Province='Ont',City='T')
```

4. EXCEPTION HANDLING

An **EXCEPTION** is an error that occurs during the execution of code. This error causes the code to raise an exception and if not prepared to handle it will halt the execution of the code: (25)

try except syntax: (26)

See some examples: (27) and (28)
Note: **except(ZeroDivisionError, NameError):** is a valid way to group together 2 exc types.

else will proove useful if you try to execute something only if there were no exc: (29)

finally allows us to always execute something even if there is an exception or not. This is usually used to signify the end of the try except: (29)

Note: If the ex is not captured within the try: except block, whatever code follows next is not considered/run.

```
(25) ZeroDivisionError: division by zero:
1/0
NameError: name 'a' is not defined:
y = a + 5
IndexError: list index out of range:
a = [1, 2, 3]
a[10]

(26)
# potential code before try catch
try:
   # code to try to execute
except:
   # code to execute if there is an exception
# code that will still execute if there is an exception

(27)
a = 1
try:
   b = int(input("Please enter a number to divide a:"))
   a = a/b
   print("Success a=",a)
except:
   print("There was an error")
print("Still executes after an exc was encountered")

# Result:
Please enter a number to divide a:
0
There was an error
Still executes after an exc was encountered

(28)
# potential code before try catch
try:
   # code to try to execute
except ZeroDivisionError:
   # code to execute if there is a ZeroDivisionError
except NameError:
   # code to execute if there is a NameError
except:
   # code to execute if ther is any exception
# code that will execute if there is no exception or one that we are
handling

(29)
# potential code before try catch
try:
   # code to try to execute
except ZeroDivisionError:
   # code to execute if there is a ZeroDivisionError
except NameError:
   # code to execute if there is a NameError
except:
   # code to execute if ther is any exception
else:
   # code to execute if there is no exception
finally:
   # code to execute at the end of the try except no matter what
# code that will execute if there is no exception or one that we are
handling
```