python_lecture_03_flow_control

February 3, 2020

1 Programming with Python

2 3 An Introduction to Flow Control

In programming languages, it is important how and in what order individual statements, instructions or function calls of a program are executed or evaluated. Therefore, a collection of different statements and functions are available to enable a control of your code flow.

2.1 Excursion: Data type bool

Boolean is data type used for logic operations. The logic values are represented by **True** and **False**. True generally used to positive or enabled situations. False is generally used negative or disabled situations. For example if we want to express existence of an item we will use boolean value.

[1]:
```
# create boolean variables
home_exists = True
car_exists = False
print('Does your home exist? -->', home_exists)
print('Does your car exist? -->', car_exists)
```

```
Does your home exist? --> True
Does your car exist? --> False
```

[2]:
```
print('home_exists is of type', type(home_exists))
print('car_exists is of type',  type(car_exists))
```

```
home_exists is of type <class 'bool'>
car_exists is of type <class 'bool'>
```

[3]:
```
# typecast boolean
print('bool(1) =', bool(1))
print('bool() =', bool())
print('bool(-1) =', bool(-1))
print('bool(10) =', bool(10))
print('bool("''") =', bool(''))
print("bool('test') =", bool('test'))
```

```
bool(1) = True
bool() = False
bool(-1) = True
bool(10) = True
bool("") = False
bool('test') = True
```

[4]:
```
# boolean expression
print('True == True -->', True == True)
print('True == False -->', True == False)
```

```
True == True --> True
True == False --> False
```

[5]:
```
var = True
print('var == True -->', var == True)
```

```
var == True --> True
```

[6]:
```
# boolean operators
number = 5

print('number == 5 -->', number == 5)
print('number < 5 -->', number < 5)
print('number <= 5 -->', number <= 5)
print('number >= 5 -->', number >= 5)
print('number != 5 -->', number != 5) # expression mark (!) stands for NOT
print('3 < number < 10 -->', 3 < number < 10)
```

```
number == 5 --> True
number < 5 --> False
number <= 5 --> True
number >= 5 --> True
number != 5 --> False
3 < number < 10 --> True
```

[7]:
```
# combining multiple boolean expressions
number = 5
print('number > 3 and number < 10 -->', number > 3 and number < 10)
print('number > 3 or number < 10 -->', number > 3 or number < 10)
```

```
number > 3 and number < 10 --> True
number > 3 or number < 10 --> True
```

2.2 3.1 if statments

if statments are used for **conditional** constructs. They perform different computations or actions depending on whether a programmer-specified boolean condition evaluates to true or false.

[8]:
```
# Check if an integer is positive
x = input("Please enter an integer: ")  # the input() function allows a␣
 ↪keyboard input. The input is always(!) read as a string.
print('input() is always a',type(x))
x = int(x) # casts string to a integer

# the if statement starts a boolean evaluation. If the evaluation is True, the␣
 ↪code underneath is executed
if x > 0:
    # code MUST be indented by four space!
    print('Integer ist positive')
```

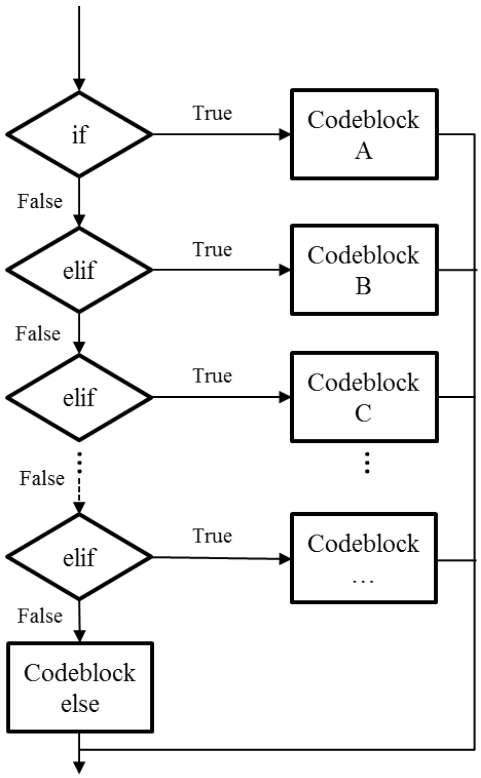
```
Please enter an integer:
input() is always a <class 'str'>


      ␣
 ↪---------------------------------------------------------------------------

       ValueError                                Traceback (most recent call␣
 ↪last)

       <ipython-input-8-3cd89c1e5f20> in <module>
         2 x = input("Please enter an integer: ")  # the input() function␣
 ↪allows a keyboard input. The input is always(!) read as a string.
         3 print('input() is always a',type(x))
   ----> 4 x = int(x) # casts string to a integer
         5
         6 # the if statement starts a boolean evaluation. If the evaluation is␣
 ↪True, the code underneath is executed


       ValueError: invalid literal for int() with base 10: ''
```

[ ]:
```
# if a condition is NOT met you can use the else statement to execute a␣
 ↪different code block
x = input("Please enter an integer: ")
x = int(x)

if x > 0:
    print('Integer is positive')

# else statement is execute if the if-statement is False
else:
    print('Integer is NOT positive')
```

[9]:
```
# you can create a series of if statement using the elif statement.
# elif is a abbreviation for "else if". The else state is always executed if␣
 ↪none of the leading if and elif statements were met.

x = int(input("Please enter an integer: "))
if x < 0:
    print('Integer is negative')
elif x > 0:
    print('Integer is positive')
else:
    print('Integer is zero')
```

```
Please enter an integer:


      ␣
↪---------------------------------------------------------------------------

       ValueError                                Traceback (most recent call␣
↪last)

       <ipython-input-9-9d84e6e206c0> in <module>
         2 # elif is a abbreviation for "else if". The else state is always␣
↪executed if none of the leading if and elif statements were met.
         3
    ----> 4 x = int(input("Please enter an integer: "))
         5 if x < 0:
         6     print('Integer is negative')


       ValueError: invalid literal for int() with base 10: ''
```

Following figure shows how the flow of a code is controlled by using if-elif-else statements.



Code flow using if-elif-else

2.3 3.2 while statements

The while statement is used if a codeblock is repeated in a **loop** until some condition is met.

[10]:
```
strings   = ['1', '2', '3', '4', '5', '6', '7', '8']
numbers   = []
squares   = []

i= 0
while i < len(strings):
    s = strings[i]
    n = int(s)
    numbers.append(n)
    squares.append(n**2)
    i = i+1

print(numbers)
print(squares)
```

```
[1, 2, 3, 4, 5, 6, 7, 8]
[1, 4, 9, 16, 25, 36, 49, 64]
```

[11]:
```
words = ['Mary', 'had', 'a', 'little', 'lamb']
i = 0
```

```
while i < len(words):
    w = words[i]
    print(i, w)
    i = i + 1
```

```
0 Mary
1 had
2 a
3 little
4 lamb
```

2.4 3.3 for statements

The for statement is used for **iterations**. In this case, Python differs a bit from C or Pascal where you always iterate over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as C). Python's for statement iterates over the items of **any sequence** (a list or a string), in the order that they appear in the sequence. For example:

[12]:
```
strings   = ['1', '2', '3', '4', '5', '6', '7', '8']
numbers   = []
squares   = []

for s in strings:
    n = int(s)
    numbers.append(n)
    squares.append(n**2)

print(numbers)
print(squares)
```

```
[1, 2, 3, 4, 5, 6, 7, 8]
[1, 4, 9, 16, 25, 36, 49, 64]
```

2.5 3.4 range() function

If you do need to iterate over a **sequence** of numbers, the built-in function range() is used. It generates arithmetic progressions

[13]:
```
# you can define a custom range it iterates
for i in range(5, 10):
    print(i)
```

```
5
6
7
8
9
```

[14]:
```
# the default step size is 1, but it can also be defined
for i in range(0, 10, 3): # here, the step size is three
    print(i)
```

```
0
3
6
9
```

[15]:
```
strings   = ['1', '2', '3', '4', '5', '6', '7', '8']
numbers   = []
squares   = []


for i in range(len(strings)):
    s = strings[i]
    n = int(s)
    numbers.append(n)
    squares.append(n**2)

print(numbers)
print(squares)
```

```
[1, 2, 3, 4, 5, 6, 7, 8]
[1, 4, 9, 16, 25, 36, 49, 64]
```

[16]:
```
# to iterate string literals over the indices of a sequence, you can combine␣
 ↪range() and len() as follows:
array = ['Mary', 'had', 'a', 'little', 'lamb']
for i in range(len(array)):
    print(i, array[i])
```

```
0 Mary
1 had
2 a
3 little
4 lamb
```

2.6 3.5 break, continue and pass

break terminates a for or while loop:

[17]:
```
array = ['OK', 'OK', 'OK', 'ERROR', 'OK', 'OK' ]
i = 0
for s in array:
    i = i + 1
    if s == 'OK':
```

```
        print(i, 'OK ', end='')
    elif s == 'ERROR':
        print(i, 'ERROR')
        break
    print('Doing something useful')
```

```
1 OK Doing something useful
2 OK Doing something useful
3 OK Doing something useful
4 ERROR
```

continue only stops the current iteration and continues with the next iteration of the same loop:

[18]:
```
array = ['OK', 'OK', 'OK', 'ERROR', 'OK', 'OK' ]
i = 0
for s in array:
    i = i + 1
    if s == 'OK':
        print(i, 'OK ', end='')
    elif s == 'ERROR':
        print(i, 'ERROR')
        continue
    print('Doing something useful')
```

```
1 OK Doing something useful
2 OK Doing something useful
3 OK Doing something useful
4 ERROR
5 OK Doing something useful
6 OK Doing something useful
```

The **pass** statement does nothing. It can be used when another statement requires syntactically some code but the program requires no action. For example:

[19]:
```
array = ['OK', 'OK', 'OK', 'ERROR', 'OK', 'OK' ]
i = 0
for s in array:
    i = i + 1
    if s == 'OK':
        print(i, 'OK ', end='')
    elif s == 'ERROR':
        print(i, 'ERROR')
        pass
    print('Doing something useful')
```

```
1 OK Doing something useful
2 OK Doing something useful
```

```
3 OK Doing something useful
4 ERROR
Doing something useful
5 OK Doing something useful
6 OK Doing something useful
```

2.7 3.6 List comprehension

One of python's more exclusive features are *list comprehensions*. A list comprehension is a short-hand notation for the generation of a new list by transforming an existing list element-wise. Using a *list comprehension*, we just start out with a an empty list literal `numbers = []` Then, we define how an element of the new list is constructed: `int(s) for s in strings` Note, that when constructin list comprehensions, it is perfectly legal to put the iteration body **in front of the normal iterator statement**. Also note, that in this case **the trailing colon (:) has to be ommitted**. Finally, we put this modified iterator statement (i.e. body in front, no trailing :) **into** the empty list literal:

[20]:
```
# converting a list of strings to integers
# classical approach
strings   = ['1', '2', '3', '4', '5', '6', '7', '8']
print('List of strings:' ,strings)

numbers = []
for s in strings:
    number = int(s)
    numbers.append(number)

print('List of numbers', numbers)
```

```
List of strings: ['1', '2', '3', '4', '5', '6', '7', '8']
List of numbers [1, 2, 3, 4, 5, 6, 7, 8]
```

[21]:
```
# converting a list of strings to integers
# list comprehension
strings   = ['1', '2', '3', '4', '5', '6', '7', '8']
print('List of strings:' ,strings)

numbers = [ int(s) for s in strings ]
squares = [ int(s)**2 for s in strings ]

print('List of numbers:', numbers)
print('List of squares:', squares)
```

```
List of strings: ['1', '2', '3', '4', '5', '6', '7', '8']
List of numbers: [1, 2, 3, 4, 5, 6, 7, 8]
List of squares: [1, 4, 9, 16, 25, 36, 49, 64]
```

[22]:
```
array = ['OK', 'OK', 'OK', 'ERROR', 'OK' ]
i = 0
```

```
for s in array:
    i = i + 1
    if s == 'OK':
        print(i, 'OK')
    elif s == 'ERROR':
        pass
```

```
1 OK
2 OK
3 OK
5 OK
```