8_bool-conditional-loops

June 1, 2023

[ ]:
```
# Boolean constants and operators.

# True and False constants.
print(type(True))
print(type(False))

# NOT operator.
print(not True)
print(not False)

# AND operator.
print(True and True)
print(True and False)
print(False and False)

# OR operator.
print(True or True)
print(True or False)
print(False or False)


# Operations returning boolean values.

# Comparison evaluating to True.
print(1 == 1)
print("a" == "a")
print(2 > 1)
print(2 >= 2)
print(2 != 1)

# Comparison evaluating to False.
print(1 == 2)
print("a" == "b")
print(2 > 3)
print(2 >= 3)
print(2 != 2)
```

```
# Conditional execution.

b = False  # set b to True or False

if b:
    print("Condition true 1.")
    print("Condition true 2.")


if b:
    print("First branch 1.")
    print("First branch 2.")
else:
    print("Second branch.")
# Note: the _indentation_ is part of the syntax.

# Multiple branches.

b1 = True
b2 = True

if b1:
    print("First branch.")
elif b2:
    print("Second branch.")
else:
    print("Third branch.")

# Nested conditionals.

b1 = False
b2 = True

if b1:
    if b2:
        print("Branch A.")
    else:
        print("Branch B.")
else:
    if b2:
        print("Branch C.")
    else:
        print("Branch D.")

# Turn (dynamic) code into (static) data.
```

```
dd = {
    (True, True): "Branch A.",
    (True, False): "Branch B.",
    (False, True): "Branch C.",
    (False, False): "Branch D."
}
# Note: dict's keys are tuples.

b1 = True
b2 = False
print(dd[(b1,b2)])  # the result is the same as the nested conditionals

# Turn the code into data reduces "spaghetti code"
# and makes it easier to debug, modify, reuse,
# understand, share the code.

#    If you love writing code - really, truly love to write
#    code - you'll love it enough to write as little of it
#    as possible.
# https://blog.codinghorror.com/the-best-code-is-no-code-at-all/


# Repeated execution: while.

counter = 0

while (counter < 3):
    print("Enter block, counter = ", counter)
    print(counter < 3)  # verify that the counter satisfies the condition
    counter = counter + 1  # increment the counter
    print("Exit block,  counter = ", counter)


# Note: the cycle
# while True:
#     print("Running.")
# will never terminate and the code will be stuck in an endless loop.


# Example: Calculate the definite integral of x^2.

xmax = 3.0
dx= 0.01

s = 0  # accumulator, starts from zero
x = 0  # counter, starts from the leftmost extreme of the integration domain
while (x < xmax):  # iterate until the rightmost extreme of the domain
```

```
    f = x**2.0  # the integrand
    s = s + f * dx  # add the area of a rectangle to the accumulator
    x = x + dx  # increase the counter by the base of the rectangle

print("Numerical result:  %8.4f" % (s))
print("Analytical result: %8.4f" % ((xmax**3.0)/3.0)) # risultato analitico␣
 ↪dell'integrale

# Note: just to demonstrate the cycle, this approach has several precision
# issues.  In particular, DO NOT add small numbers repeatedly or numerical
# error accumulates.

# Example: Calculate a multiplication table.

dd = {}  # empty dictionary

i1 = 1  # initialize the counter
while (i1 <= 10):
    i2 = 1  # initialize the counter
    while (i2 <= 10):  # _nested_ while loop
        dd[(i1,i2)] = i1 * i2
        i2 = i2 + 1
    i1 = i1 + 1

s = ""  # empty string

i1 = 1
while (i1 <= 10):
    i2 = 1
    while (i2 <= 10):
        if (i2 >= i1):
            s = s + "%3d" % (dd[(i1,i2)])
        else:
            s = s + "   "
        i2 = i2 + 1
    s = s + "\n"
    i1 = i1 + 1

print(s)


# Execution over iterables: for.



# List.
```

```python
nn = [4, 3, 7]

for i in nn:
    print(i)

# Dictionary.

dd = {"a": 1, "b": 2, "c": 3}

for key in dd:
    print(key, dd[key])

# Output of the function range.

for i in range(5):
    print(i)

for i in range(2, 9, 3): #da 2 a 9 di 3 in 3
    print(i)
# Note: use Numpy's arange function for large iterations.


# Output of the function enumerate.

ss = ["c", "b", "a"]

for i,s in enumerate(ss):
    print(i, s) #stampa l'indice e il valore


# Output of the function zip.

xx = ["a", "b", "c"]
yy = [1, 2, 3]

for x,y in zip(xx,yy):
    print(x,y) #stampa i valori in posizione i-esima

# Note: try to do loops in a "pythonic" way.
# Example: less pythonic way to accomplish the same task.

xx = ["a", "b", "c"]
yy = [1, 2, 3]

for i in range(len(xx)):
    print(xx[i],yy[i])
```

```python
# List comprehensions.

xx = [i for i in range(5)]
print(xx)

xx = [2*i*i for i in range(5)]
print(xx)

xx = [i for i in range(5) if i != 2] #se i è diverso da 2
print(xx)


# Interrupt a cycle.

# Break a while.

i = 0
while True: #ciclo infinito - tautologia
    print(i)
    if (i > 3):
        break
    i = i + 1
#prima dei due punti c'è una condizione che viene valutata come vera o falsa;␣
 ↪se vera, si entra nel ciclo

# Break a for.

xx = ["a", "b", "c", "d", "e"]

for x in xx:
    if (x == "d"):
        break
    print(x)

# Continue a while.

i = 0
while (i < 5):
    if (i == 3):
        i = i + 1
        continue
    print(i)
    i = i + 1


# Continue a for.
```

```
xx = ["a", "b", "c", "d", "e"]

for x in xx:
    if (x == "d"):
        continue
    print(x)


# Note: these command might produce spaghetti code.  But sometime they might
# spare many, many lines of otherwise convoluted logic.  Think.
# With great power comes great responsibility.
```

```
<class 'bool'>
<class 'bool'>
False
True
True
False
False
True
True
False
True
True
True
True
True
False
False
False
False
False
Second branch.
First branch.
Branch C.
Branch B.
Enter block, counter =  0
True
Exit block,  counter =  1
Enter block, counter =  1
True
Exit block,  counter =  2
Enter block, counter =  2
True
Exit block,  counter =  3
Numerical result:    9.0450
Analytical result:   9.0000
```

```
  1  2  3  4  5  6  7  8  9 10
     4  6  8 10 12 14 16 18 20
        9 12 15 18 21 24 27 30
          16 20 24 28 32 36 40
             25 30 35 40 45 50
                36 42 48 54 60
                   49 56 63 70
                      64 72 80
                         81 90
                           100
```

```
4
3
7
a 1
b 2
c 3
0
1
2
3
4
2
5
8
0 c
1 b
2 a
a 1
b 2
c 3
a 1
b 2
c 3
[0, 1, 2, 3, 4]
[0, 2, 8, 18, 32]
[0, 1, 3, 4]
0
1
2
3
4
a
b
c
0
1
2
```

```
4
a
b
c
e
```

RIASSUNTO:

1. VARIABILI BOOLEANE (TRUE, FALSE)
   - True e False sono della classe “bool”
   - i NOT operatore rende True False e False True
   - l’AND sfrutta le tavole di verità: T T -> T; T F -> F; F F -> F
   - l’OR sfrutta le tavole di verità: T T -> T; T F -> T; F F -> F
   - i valori di True e False si possono ottenere come risultato di affermazioni sempre vere o false
2. CONDIZIONALI (IF, ELIF, ELSE)
   - settando delle variabili su vere o false, si possono scrivere degli if elif else che stampano stringhe
   - si possono scrivere i condizionali innestati, cioè if dentro altri if, che si possono trasformare da codice in dati tramite definizione di dizionario dd = {} i cui elementi sono tuple di valori booleani che restituiscono un risultato a seconda di quale è l’operazione di dd che si definisce dopo e a seconda dei valori booleani che si assegnano alle variabile dell’operazione del dd
3. CICLI RIPETUTI (WHILE)
   - permette di descrivere un’azione finché viene rispettata una condizione
   - permette di calcolare un integrale
   - permette di scrivere una tabellina moltiplicativa
   - è possibile interrompere o continuare un while
4. CICLI ITERATI SU SEQUENZE (FOR)
   - possono essere utilizzati su liste e dizionari
   - la funzione range prende o lo stop, oppure lo start, stop e il numero di passi
   - la funzione enumerate stampa l’indice e il valore
   - la funzione zip stampa il valore e l’indice (modo veloce invece che usare due liste su range)
   - si possono usare come parte integrante di liste: e.g. al posto di scrivere tutti i valori, si usa un for
   - è possibile interrompere un for data una certa condizione o di continuare (ossia riprendere senza eseguire il comando finale)