02 Control statements

September 13, 2020

1 Introduction

Control statements are central to computing. They allow a program to change what it does depending on input or other data. Typical flows in a computer program involve structures like:

- if 'X' do task 'A', else if 'Y' do task 'B'
- perform the task 'A' 'N' times
- perform the task 'B' until 'X' is true

Flows like these are implemented using 'control statements'. Control statements are a major part of all non-trivial computer programs.

1.1 Objectives

- Introduce Boolean types
- Introduce comparison operators
- Learn to use control statements

1.2 Example of a control statement in pseudo code

An electric window opener, attached to a rain sensor and a temperature gauge, might be controlled by the following program:

```
if raining:  # If raining, close the window
    close_window()
else if temperature > 26:  # If the temperature is over 26 deg, open window
    open_window()
else if temperature < 19:  # If the temperature is below 19 deg, close window
    close_window()
else:  # Otherwise, do nothing and leave window as it is
    pass
```

It is easy to imagine the program being made more sophisticated using the time of the day and the day of the week, or being attached to a smoke alarm.

We will look at different types of control statements, but first we need to introduce boolean types and comparison operators.

2 Booleans

Before starting with control statements, we need to introduce booleans. A Boolean is a type of variable that can take on one of two values - true or false.

[ ]:
```
a = True
print(a)

a = False
print(a)
```

Booleans are used extensively in control statements.

3 Comparison operators

We often want to check in a program how two variables are related to each other, for example if one is less than the other, or if two variables are equal. We do this with 'comparison operators', such as `<`, `<=`, `>`, `>=` and `==`. These are examples of *binary operators*.

Below is an example checking if a number `a` is less than or greater than a number `b`:

[ ]:
```
a = 10.0
b = 9.9
print(a < b)
print(a > b)
```

Equality is checked using '`==`', and '`!=`' is used to test if two variables are not equal. Below are some examples to read through.

[ ]:
```
a = 14
b = -9
c = 14

# Check if a is equal to b
print("Is a equal to b?")
print(a == b)

# Check if a is equal to c
print("Is a equal to c?")
print(a == c)

# Check if a is not equal to c
print("Is a not equal to c?")
print(a != c)

# Check if a is less than or equal to b
print("Is a less than or equal to b?")
print(a <= b)
```

```
# Check if a is less than or equal to c
print("Is a less than or equal to c?")
print(a <= c)

# Check if two colours are the same
colour0 = 'blue'
colour1 = 'green'
print("Is colour0 the same as colour1?")
print(colour0 == colour1)
```

4 Boolean operators

In the above we have only used one comparison at a time. Boolean operators allow us to 'string' together multiple checks using the operators '`and`', '`or`' and '`not`'. The operators '`and`' and '`or`' take a boolean on either side, and the code

```
X and Y
```

will evaluate to `True` if `X` *and* `Y` are both true, and otherwise will evaluate to `False`. The code

```
X or Y
```

will evaluate to `True` if `X` *or* `Y` is true, and otherwise will evaluate to `False`. Here are some examples:

[ ]:
```
# If 10 < 9 (false) and 15 < 20 (true) -> false
print(10 < 9 and 15 < 20)
```

[ ]:
```
# Check if 10 < 9 (false) or 15 < 20 (true) -> true
print(10 < 9 or 15 < 20)
```

The meaning of the statement becomes clear if read left-to-right.

Below is a very simple example that, given the current time of day reports

- true if it is lunch time; and
- true if we are outside of working hours.

[ ]:
```
time = 13.05  # The current time

work_starts = 8.00  # Start of working day
work_ends = 17.00  # End of working day

lunch_starts = 13.00  # Start of lunchtime
lunch_ends = 14.00  # End of lunchtime

# Check if it's lunch time
print("Is it lunchtime?")
is_lunchtime = time >= lunch_starts and time < lunch_ends
print(is_lunchtime)
```

```
# Check if we're outside of working hours
print("Are we outside of working hours?")
outside_working_hours = time < work_starts or time >= work_ends
print(outside_working_hours)
```

Note that the comparison operators (`>=`, `<=`, `<` and `>`) are evaluated before the Boolean operators (`and`, `or`).

In Python, the '`not`' operator negates a statement, e.g.:

[ ]:
```
# Is 12 *not* less than 7 -> true
a = 12
b = 7
print(not a < b)
```

Only use '`not`' when it makes a program easy to read. For example,

[ ]:
```
print(not 12 == 7)
```

is not good practice. Better is

[ ]:
```
print(12 != 7)
```

Here is a double-negation, which is very cryptic (and poor programming):

[ ]:
```
print(not not 12 == 7)
```

4.1 Multiple comparison operators

The examples so far use at most two comparison operators. In some cases we might want to perform more checks. We can control the order of evaluation using brackets. For example, if we want to check if a number is strictly between 100 and 200, or between 10 and 50:

[ ]:
```
value = 150.5
print ((value > 100 and value < 200) or (value > 10 and value < 50))
```

The two checks in the brackets are evaluated first (each evaluates to `True` or `False`), and then the '`or`' checks if one of the two is true.

5 Control statements

Now that we've covered comparisons, we are ready to look at control statements. Here is a control statement in pseudo code:

```
if A is true
    Perform task X (only)
else if B is true
    Perform task Y (only)
else
```

```
    Perform task Z (only)
```

The above is an 'if-else' statement. Another type of control statement is

```
do task X 10 times
```

We make this concrete below with some examples.

5.1 `if` statements

Below is a simple example that demonstrates the Python syntax for an if-else control statement. For a value assigned to a variable `x`, the program prints a message and modifies `x`. The message and the modification of `x` depend on the initial value of `x`:

[ ]:
```
x = -10.0  # Initial x value

if x > 0.0:
    print('Initial x is greater than zero')
    x -= 20.0
elif x < 0.0:
    print('Initial x is less than zero')
    x += 21.0
else:
    print('Initial x is not less than zero and not greater than zero, therefore␣
 ↪it must be zero')
    x *= 2.5

# Print new x value
print("New x value:", x)
```

Try changing the value of `x` and re-running the cell to see the different paths the code can follow.

We now dissect the control statement example. The control statement begins with an `if`, followed by the expression to check, followed by ':'

```
if x > 0.0:
```

Below that is a block of code, indented by four spaces, that is executed if the check `x > 0.0` is true:

```
    print('Initial x is greater than zero')
    x -= 20.0
```

and in which case the program will then move beyond the end of the control statement. If the check evaluates to false, then the `elif` (else if) check

```
elif x < 0.0:
    print('Initial x is less than zero')
    x += 21.0
```

is performed, and if true `print('x is less than zero')` is executed and the control block is exited. The code following the `else` statement is executed

```
else:
    print('Initial x is not less than zero and not greater than zero, therefore it must be zero
```

if none of the preceding statements were true.

5.1.1 Example: currency trading

A currency trader makes a commission by selling US dollars to travellers at a rate below the market rate. The mark-down multiplier they apply is show below.

Amount (GBP)	reduction on market rate
Less than 100	0.9
From 100 and less than 1,000	0.925
From 1,000 and less than 10,000	0.95
From 10,000 and less than 100,000	0.97
Over 100,000	0.98

The currency trader incurs extra costs for handling cash over electronic transactions, so for cash transactions they retain an extra 10% after conversion.

At the current market rate 1 GBP is 1.25607 USD.

[ ]:
```
GBP  = 15600.05  # The amount in GBP to be changed into USD
cash = True  # True if selling cash, otherwise False

market_rate = 1.25607  # 1 GBP is worth this many dollars at the market rate

# Apply the appropriate reduction depending on the amount being sold
if GBP < 100:
    USD = 0.9*market_rate*GBP
elif GBP < 1000:
    USD = 0.925*market_rate*GBP
elif GBP < 10000:
    USD = 0.95*market_rate*GBP
elif GBP < 100000:
    USD = 0.97*market_rate*GBP
else:
    USD = 0.98*market_rate*GBP

if cash:
    USD *= 0.9  # recall that this is shorthand for USD = 0.9*USD

print("Amount in GBP sold:", GBP)
print("Amount in USD purchased:", USD)
print("Effective rate:", USD/GBP)
```

5.2 `for` loops

A `for` loop is a block that repeats an operation a specified number of times (loops). The concept is rich, but we start with the simplest and most common usage:

[ ]:
```
for n in range(4):
    print("----")
    print(n, n**2)
```

The above executes the loop 4 times over the integers 0, 1, 2 and 3. The statement

```
for n in range(4):
```

says that we want to loop over four integers, and by default it starts from zero (see https://docs.python.org/3/library/stdtypes.html#range for the documentation for `range`). The value of `n` is incremented in each loop iteration. The code we want to execute inside the loop is indented by four spaces:

```
    print("----")
    print(n, n**2)
```

The loop starts from zero and does not include 4 - `range(4)` is a shortcut for `range(0, 4)`. We can change the starting value if we need to:

[ ]:
```
for i in range(-2, 3):
    print(i)
```

The loop starts at -2, but does not include 3. If we want to increment in steps of three rather than one:

[ ]:
```
for n in range(0, 10, 3):
    print(n)
```

5.2.1 Example: conversion table from degrees Fahrenheit to degrees Celsius

We can use a `for` loop to create a conversion table from degrees Fahrenheit (T_F) to degrees Celsius (T_c), using the formula:

$$T_c = 5(T_f - 32)/9$$

Computing the conversion from -100 F to 200 F in steps of 20 F (not including 200 F):

[ ]:
```
print("T_f,    T_c")
for Tf in range(-100, 200, 20):
    print(Tf, (Tf - 32)*5/9)
```

5.3 `while` loops

We have seen that `for` loops execute the loop body a specified number of times. A `while` loop performs a task while a specified statement is true. For example:

[ ]:
```
print("Start of while statement")
x = -2
while x < 5:
    print(x)
    x += 1  # Increment x
print("End of while statement")
```

The body of the `while` statement, which follows the `while` statement and is indented four spaces, is executed and repeated until `x < 5` is `False`.

It can be quite easy to cause a crash using a `while` loop. E.g.,

```
x = -2
while x < 5:
    print(x)
```

will continue indefinitely since `x < 5 == False` will never be satisfied. This is known as an *infinite loop*. It is usually good practice to add checks to avoid an infinite loop, e.g. specify a maximum number of permitted loops. More on avoiding infinite loops below.

The above example could have been implemented using a `for` loop, and a `for` loop would be preferred in this case. The following is an example of where a `while` is appropriate:

[19]:
```
x = 0.9
while x > 0.001:
    # Square x (we could have used the shorthand x *= x)
    x = x*x
    print(x)
```

```
0.81
0.6561000000000001
0.43046721000000016
0.18530201888518424
0.03433683820292518
0.001179018457773862
1.390084523771456e-06
```

since we might not know beforehand how many steps are required before `x > 0.001` becomes false.

If $x \geq 1$, the above would lead to an infinite loop. To make a code robust, it would be good practice to check that $x < 1$ before entering the `while` loop.

5.4 `break`, `continue` and `pass`

5.4.1 `break`

Sometimes we want to break out of a `for` or `while` loop. Maybe in a `for` loop we can check if something is true, and then exit the loop prematurely, e.g.

[20]:
```
for x in range(10):
    print(x)
```

```
    if x == 5:
        print("Time to break out")
        break
```

```
0
1
2
3
4
5
Time to break out
```

Below is a program for finding prime numbers that uses a **break** statement. Take some time to understand what it does. It might be helpful to add some print statements to understand the flow.

[21]:
```
N = 50  # Check numbers up 50 for primes (excludes 50)

# Loop over all numbers from 2 to 50 (excluding 50)
for n in range(2, N):

    # Assume that n is prime
    n_is_prime = True

    # Check if n can be divided by m, where m ranges from 2 to n (excluding n)
    for m in range(2, n):
         if n % m == 0:  # This is true if the remainder for n/m is equal to␣
↪zero
            # We've found that n is divisable by m, so it can't be a prime␣
↪number.
            # No need to check for more values of m, so set n_is_prime = False␣
↪and
            # exit the 'm' loop.
            n_is_prime = False
            break

    #  If n is prime, print to screen
    if n_is_prime:
        print(n)
```

```
2
3
5
7
11
13
17
19
23
```

```
29
31
37
41
43
47
```

Try modifying the code for finding prime numbers such that it finds the first N prime numbers (since you do not know how many numbers you need to check to find N primes, use a `while` loop).

5.4.2 `continue`

Sometimes we want to go prematurely to the next iteration in a loop, skipping the remaining code in the loop body. For this we use `continue`. Here is an example that loops over 20 numbers (0 to 19) and checks if the number is divisible by 4. If it is divisible by 4 it prints a message before moving to the next value. If it is not divisible by 4 it advances the loop.

[22]:
```
for j in range(20):
    if j % 4 == 0:  # Check remained of j/4
        continue  # jump to next iteration over j
    print("Number is not divisible by 4:", j)
```

```
Number is not divisible by 4: 1
Number is not divisible by 4: 2
Number is not divisible by 4: 3
Number is not divisible by 4: 5
Number is not divisible by 4: 6
Number is not divisible by 4: 7
Number is not divisible by 4: 9
Number is not divisible by 4: 10
Number is not divisible by 4: 11
Number is not divisible by 4: 13
Number is not divisible by 4: 14
Number is not divisible by 4: 15
Number is not divisible by 4: 17
Number is not divisible by 4: 18
Number is not divisible by 4: 19
```

5.4.3 `pass`

Sometimes we need a statement that does nothing. It is often used during development where syntactically some code is required but which you have not yet written. For example:

[23]:
```
for x in range(10):
    if x < 5:
        # TODO: implement handling of x < 5 when other cases finished
        pass
    elif x < 9:
        print(x*x)
```

```
    else:
        print(x)
```

```
25
36
49
64
9
```

It can also help readability. Maybe in a program nothing needs to be done for specific cases, but someone reading the code might reasonably think that something should be done and suspect a bug. Using `pass` says to the reader that it was the programmer's intention that nothing should be done.

5.5 Infinite loops: cause and guarding against

A common bug, especially when using `while` statements, is the infinite loop. This is when a loop is entered but never terminates (exits). Infinite loops can render a system unresponsive, sometimes requiring a shutdown to restore function.

It is good practice, espeically when learning, to add guards against infinite loops. For example,

[24]:
```
x = 0.0

counter = 0
while x < 0.05:

    # Guard against infinite loop
    counter += 1
    if counter > 2000:
        print("Loop count exceeded 2000. Exiting")
        break
```

```
Loop count exceeded 2000. Exiting
```

6 Exercises

Complete now the 02 Exercises notebook.