# Chapter 3

# Control Structures

## 3.1 The `for` loop

Control structures allow sections of your program to be repeated or skipped when certain conditions are met. One of the most useful control structures is the `for` loop. Here is the syntax:

`for` *variable* `in` *list*:
    *code to execute*

This is best illustrated with a simple example:

```
for x in [1.2, 3.5, "cat"]:
    print(x)
```

Here, the *variable* is `x` and the *list* is `[1.2, 3.5, "cat"]`. The *code to execute* is `print(x)`. The `for` statement tells Python to execute the indented code for each value of the variable `x`. The *code to execute* is repeated three times, first, for `x = 1.2`, then for `x = 3.5`, then for `x = "cat"`.

Note that the `for` statement always ends in a colon and *code to execute* is indented. The indentation can be any number of spaces, or the tab character. If the *code to execute* requires multiple commands, then each line must be indented. Any unindented statements that follow the indented lines of code will not be included in the `for` loop.

**Exercise 3.1a**

Run the code above. Make sure to indent the print statement. Add other numbers or strings to the list. Add one more line, such as `print("wow!")`, to the end of the code. What happens if this last command is indented? Not indented?

Here is an example that uses a `for` loop to calculate 4!, the factorial of the number 4:

```
fac = 1              # assign the variable fac to 1

for i in [2, 3, 4]:  # repeat indented code for i = 2, 3, 4
    fac = i*fac

print(f"4! = {fac})  # print the result
```

The `for` statement causes the indented line to repeat for each of the three values of the variable `i` that appear in the list `[2, 3, 4]`. This code is logically equivalent to

```
fac = 1              # assign the variable fac to 1

i = 2                # set i = 2
fac = i*fac          # reassign fac to 2*1 = 2

i = 3                # set i = 3
fac = i*fac          # reassign fac to 3*2 = 6

i = 4                # set i = 4
fac = i*fac          # reassign fac to 4*6 = 24

print(f"4! = {fac}")  # print the result
```

**Exercise 3.1b**

The "double factorial" is defined by

$$n!! = \begin{cases} n \cdot (n-2) \cdot \cdots \cdot 3 \cdot 1 \ , & n \text{ odd}, \\ n \cdot (n-2) \cdot \cdots \cdot 4 \cdot 2 \ , & n \text{ even}. \end{cases}$$

Write a program that uses a `for` loop to compute 13!!.

Note that indentation must be consistent throughout your program. Whether you choose a certain number of spaces, or the tab character, you must stick with that choice. You cannot mix indentation types within a single code. It does not matter if comments are indented.

**Exercise 3.1c**

Write a code to compute the square roots of 3, 6, 9, 12 and 15. Use a `for` loop and a list. Print the results to 4 decimal places.

### 3.2 The `range()` function

In the last section we computed 4! using a `for` loop along with the list `[2, 3, 4]`. How can we modify our code to compute 97!? It would be tedious to create a list of integers from 2 through 97. This problem is solved with the `range()` function:

```
fac = 1               # assign the variable fac to 1

for i in range(2,98): # repeat indented code for i = 2,...,97
    fac = i*fac

print(f"97! = {fac})  # print the result
```

The command `range(a,b)` creates a Python list of integers beginning with `a`, and ending with `b-1`. That's right—the last number in the list is `b-1`. The number `b` in `range(a,b)` is the first integer that is *excluded* from the list. In the code above, `range(2,98)` produces the list of integers $2, \ldots, 97$.

With the `range()` function, we can easily modify our code to compute the factorial of any positive integer chosen by the user:

```
# input number and convert the string to an integer
n = input("input a positive integer:")
n = int(n)

# set fac to 1 then repeat indented code for i = 2,...,n
fac = 1
for i in range(2,n+1):
    fac = i*fac

print(f"{n}! = {fac})
```

**Exercise 3.2a**

Write a program that asks the user to input two integers, $n$ and $m$, with $n > m$. The code should compute the product

$$m \cdot (m+1) \cdots (n-1) \cdot n$$

(This is the product of all integers beginning with $m$ and ending with $n$.) Test your code with small input values to make sure it works correctly.

**Exercise 3.2b**

Modify the projectile code from Sec. 2.7 to compute the projectile's position at times $t = 1, 2, \ldots, 5$.

### 3.3 More about `range()`

The `range()` function produces a list of integers.[1] We can use this function in various ways. For integers `a`, `b`, `s`, and `n`,

- `range(a,b)` creates a list of integers beginning with `a`, incrementing by 1, and ending *before* `b` (that is, ending with `b-1`).
- `range(a,b,s)` creates a list of integers beginning with `a`, incrementing by `s`, and ending *before* `b`.
- `range(n)` creates a list of $n$ integers beginning with 0, incrementing by 1, and ending *before* `n` (that is, ending with `n-1`).

**Exercise 3.3a**

Run this code:

```
for i in range(-2,5):
    print(f"i = {i}")

for j in range(2,8,3):
    print(f"j = {j}")

for k in range(4):
    print(f"k = {k}")
```

Pay attention to the first and last numbers in each list!

**Exercise 3.3b**

Write a program that asks the user to input an odd integer $n$, then computes $n!!$. Test your code with small input values to make sure it works correctly.

[1] Strickly speaking, `range()` creates an "immutable sequence," not a list. (Immutable means the elements cannot be modified.) An immutable sequence behaves like a list when it appears in a `for` loop.

The following code uses a `for` loop and `range()` to compute

$$\sum_{n=1}^{10} n \ ,$$

the sum of the first 10 positive integers:

```
sum = 0
for n in range(1,11):
    sum = sum + n
print("sum = ", sum)
```

Note that the variable `sum` is given an initial value of `0` before entering the loop.

**Exercise 3.3c**

Write a code to compute

$$S = \sum_{k=1}^{N} k = 1 + 2 + 3 + \cdots + N$$

for a given integer $N$. Experiment with different values of $N$ to find the smallest number $N$ such that $S > 10^6$ .

### 3.4 Conditions

A *condition* is a statement that is either `True` or `False`. Most conditions involve a comparison between two quantities. For example, the condition `5 > 8` is `False`. You can use the following operators for comparison:

`==` (equivalence, equal to)
`!=` (not equal to)
`<` (less than)
`>` (greater than)
`<=` (less than or equal to)
`>=` (greater than or equal to)

**Exercise 3.4a**

Predict the output from this code:

```
x = 3 + 2
print(x == 5)
y = 4 + 5
```

```
print(y == "cat")
```

Now run the code. Were your predictions correct?

Remember: The symbol `=` is an "assignment operator" and `==` is a comparison operator.

**Exercise 3.4b**

Evaluate these conditionals:

- `0 == 5`
- `0 <= 5`
- `0 > 5`
- `6 != 3`
- `3 == 3`
- `6 != 6`
- `2 < 2`
- `2 >= 2`

### 3.5 `if`, `elif`, `else`

Conditions can be used with the `if`, `elif`, `else` construct to control the logical flow of a code. The keyword `elif` is short for "else if."

The `if` statement can be used alone:

`if` *condition*:
    *code to execute if the condition evaluates to* `True`

If *condition* evaluates to `True`, then Python executes the indented line (or lines) of code below the `if` statement. If *condition* evaluates to `False`, Python skips the indented line (or lines) of code.

The `if` statement can be used with `else`:

`if` *condition*:
    *code to execute if the condition evaluates to* `True`
`else`:
    *code to execute if the condition evaluates to* `False`

The `if` statement can also be used with `elif` and `else`:

`if` *condition 1*:
    *code to execute if condition 1 evaluates to* `True`

`elif` *condition 2*:
    *code to execute if condition 1 evaluates to* `False` *and condition 2 evaluates to* `True`
`else`:
    *code to execute if both condition 1 and condition 2 evaluate to* `False`

Notice that, just as with a `for` statement, the blocks of code to be executed are indented. Also there is a colon `:` following each of the `if`, `elif` and `else` statements. Observe:

- You can extend the `if` construction to include as many `elif` branches as you like.
- You can have only one `if` branch and one `else` branch.
- The `elif` and `else` branches are optional.

Let's assume the user has been asked to input two numbers, $M$ and $N$. The code

```
if M > N:
    print("M is greater than N")
```

will check to see if the number $M$ is greater than $N$.

**Exercise 3.5a**

Extend this code to determine whether $M > N$, $M < N$ or $M = N$ by using `if` and two `elif` branches. Modify your code to use `if` with one `elif` branch and one `else` branch.

`if` statements can appear anywhere in a code, including the middle of a loop. In the example below, the `for` loop is executed 10 times, once for each value of $n$ from 1 through 10. Inside the loop the `if` and `else` statements check to see if the numbers $n$ are even or odd. Recall that `n%2` gives the remainder of $n$ divided by 2.

```
for n in range(1,11):
    if n % 2 == 0:        # n is even
        print(n,'is even')
    else:                 # n is odd
        print(n,'is odd')
```

Observe that the code to execute under the `for` statement is indented. The codes to execute under the `if` and `else` statements are "doubly indented."

**Exercise 3.5b**

Write a program that asks the user to input any positive integer $n$ (even or odd) then computes $n!!$.

**Exercise 3.5c**

Write a code that will examine every integer $n$ from 1 through 100. If the integer is divisible by 7, the code should print $n$ and $n^2$. If the number is divisible by 11, the code should print $n$ and $\sqrt{n}$.

### 3.6 `and`, `or`, `not`

Sometimes more than one condition needs to be satisfied. The expression

*condition 1* `and` *condition 2*

will evaluate to `True` if both *condition 1* and *condition 2* are `True`. The expression

*condition 1* `or` *condition 2*

will evaluate to `True` if either *condition 1* or *condition 2* (or both) are `True`. The command

`not` *condition*

will evaluate to `True` if *condition* evaluates to `False`.

As an example, you might like to know which numbers from 1 through 20 are divisible by 2 and by 3:

```
for n in range(1,21):
    if n%2 == 0 and n%3 == 0:
        print(n, 'is divisible by two and by three')
```

**Exercise 3.6**

Find the numbers 1 through 20 that are divisible by either 2 or 3 or both.

### 3.7 `continue` and `break`

The keywords `continue` and `break` are used to alter the flow of the code through a `for` loop. These keywords can also be used with `while` loops; see Sec. 7.6.

Imagine a code that asks the user to choose a positive integer $N$. The code then computes the squares of each interger from 1 through $N$. After the integer has been input, the code looks like this:

```
for i in range(1,N+1):
    isqr = i*i
    print(i,'*',i,'=',isqr)
print('the end')
```

For $N = 4$, the output is

```
1 * 1 = 1
2 * 2 = 4
3 * 3 = 9
4 * 4 = 16
the end
```

Now imagine a code that asks the user to input two positive integers, $N$ and $M$. The code computes the squares of each integer from 1 through $N$, but skips $M$. Modify the code above by adding `continue` inside an `if` statement:

```
for i in range(1,N+1):
    if i == M:
        continue
    isqr = i*i
    print(i,'*',i,'=',isqr)
print('the end')
```

For $N = 4$ and $M = 3$, the output is

```
1 * 1 = 1
2 * 2 = 4
4 * 4 = 16
the end
```

The `continue` keyword interrupts the normal cycle of the `for` loop. As shown in Fig. 3.1, `continue` stops the current cycle through the loop and instructs the code to continue with the next cycle.

Finally, imagine a code that asks the user to input positive integers $N$ and $M$. The code computes the squares of each integer from 1 through $N$, but skips $M$ and all integers larger than $M$. If we could be sure that $M \leq N$, we could simply change the range of the `for` loop to `range(1,M+1)`. But this won't work if $M > N$. Instead, we can modify the previous code by replacing `continue` with `break`:

```
for i in range(1,N+1):
    if i == M:
        break
    isqr = i*i
```

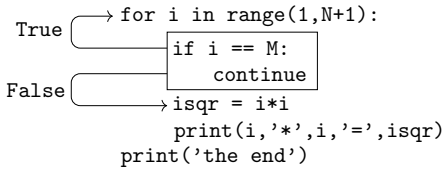


Fig. 3.1: When the condition `i == M` evaluates to `True`, the `continue` keyword tells the code to continue with the next cycle of the `for` loop.

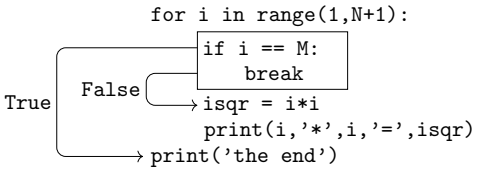


Fig. 3.2: When the condition `i == M` evaluates to `True`, the `break` keyword tells the code to stop cycling through the `for` loop.

```
    print(i,'*',i,'=',isqr)
print('the end')
```

For $N = 4$ and $M = 3$, the output is

```
1 * 1 = 1
2 * 2 = 4
the end
```

The `break` keyword also interupts the normal cycle of the `for` loop. As shown in Fig. 3.2, `break` instructs the code to break away from the `for` loop and move to the next line of code outside the loop.

**Exercise 3.7a**

Create a code that asks the user to input three positive integers, $N$, $M_1$ and $M_2$. The code should compute (and print out) the squares of all integers 1 through $N$, but skip $M_1$ and stop computing if it reaches $M_2$.

**Exercise 3.7b**

Write a Python program to compute the sum of all positive integers 1 through 100 that are divisible by 3 or by 5, but not both.

### 3.8 More exercises

**Exercise 3.8a**

It is well known that the geometric series

$$\sum_{n=0}^{\infty} 1/2^n$$

is equal to 2. Verify this by computing the partial sum $\sum_{n=0}^{N} 1/2^n$ and showing that the result tends to 2 as $N$ increases.

**Exercise 3.8b**

Carry out numerical experiments to approximate the value of the Riemann zeta function

$$\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s} \ ,$$

for $s = 2$, 3 and 4. Compare with the known results $\zeta(2) = \pi^2/6$ and $\zeta(4) = \pi^4/90$. There is no closed form expression for $\zeta(3)$.

**Exercise 3.8c**

The Leibniz formula for $\pi$,

$$\pi = 4 \sum_{k=1}^{\infty} \frac{(-1)^{k+1}}{2k-1} \ ,$$

is obtained by evaluating the Taylor series for $\tan^{-1} x$ at $x = 1$. Approximate the infinite series by computing partial sums. How many terms are required for the answer to match $\pi$ to 6 decimal places?