Contents

1 Iteration

Iteration is a fundamental concept in programming that allows you to execute an action repeatedly, either a *fixed number of times* or *until a certain condition is met*. There are two types of loops, which correspond to how they terminate: *counter-controlled* loops and *sentinel-controlled loops*.

1.1 Counter-controlled loops

Counter-controlled loops are used when you know how many iterations are needed *before starting*. For example, to find the average height of a class, you need to measure the height of every student in the class - a countable known, and fixed quantity.

1.2 Sentinel-controlled loops

Sentinel-controlled loops are used when the number of iterations needed is *not known beforehand*. For example, to find the total earnings in a day of the mixed rice stall in South Spine, you need to measure how much every customer of the stall spends, until the stall closes - an unknown, variable quantity that changes from day to day. There is, however, a terminating condition, which is when the stall closes.

2 Iteration in Python

To utilise iteration in Python, there are two keywords which we can use: `while` and `for`. Both can be used either in a counter-controlled or sentinel-controlled manner, though the form of each lends itself to one type or the other.

In both `while` and `for` loops, the code to be repeated (iterated) over is indented, similar to conditional `if` blocks.

2.1 `while` loops

`while` loops are more suited for sentinel-controlled loops, but can be used as counter-controlled loops.

```python
while (condition is True):
  print("looping")
  # do stuff
```

When Python first encounters a `while` loop, it checks if the condition specified is true. If the condition is true, then it proceeds to run the indented block after the `while` statement. Once it reaches the end of the indented block, Python then evaluates whether the condition specified is true again. If the condition is false, then it will not run the indented block, and skip to the next line after the indented block ends. The indented block will continue to be executed as long as the condition specified is true.

For example, in the `while` loop below, each loop increases `i` by 1, and will keep running until `i` is 3 or higher. Run the code below and reason to yourself which lines it runs.

```python
i = 0
while i < 3:
  print("i =", i, end=".")
  i += 1
  print("i increased. i =", i)
print("while loop exited")
```

It is possible, and extremely easy to get stuck in an *infinite loop*. Setting the condition to be `while True` will cause an infinite loop, until a `break` statement (described below) is encountered.

2.2 The `range()` function

Before moving on to `for` loops, we must first learn about the `range()` function, which is frequently used in conjunction with it. The `range()` function simply generates a range of numbers, and comes in several forms:

2.2.1 `range(end)`

When passed a single parameter, the `range()` function generates the range of integers from 0 to $\text{end} - 1$. i.e. if `range(n)`, the generated range is $\{0, 1, 2, \ldots, n-2, n-1\}$, for a total of n numbers. *The range function is always exclusive of the ending number.*

```python
print(range(5))       # will just display range(5)
# to print the contents of the range, we will convert it into a list.
print(list(range(5))) # [0, 1, 2, 3, 4]
```

2.2.2 `range(start, end)`

When passed two parameters, the `range()` function generates the range of integers from start to $\text{end} - 1$. i.e. if `range(p, q)` with $p < q$, the generated range is $\{p, p+1, \ldots, q-2, q-1\}$, for a total of $q - p$ numbers.

```python
print(list(range(3, 9))) # [3, 4, 5, 6, 7, 8]
```

2.2.3 `range(start, end, step)`

When three parameters are passed, the `range()` function generates integers between start to $\text{end} - 1$ with a difference of step between each adjacent pair. The set notation for this case is more complicated and is omitted. Basically, every number starting at `start` and increasing in multiples of `step`, up until but not including `end`. Note that you can also use negative `step` values.

```
print(list(range(3, 9, 2)))  # [3, 5, 7]
print(list(range(9, 3, -2))) # [9, 7, 5]
```

The other two forms of the `range()` function are special cases of this three parameter form, just with `start = 0` and `step = 1`.

2.3 for loops

The structure of **for** loops naturally lend itself to counter-controlled loops.

```
for variable in iterable:
  print("looping")
  # do stuff
```

The loop will repeat over every value in the *iterable*, until there are no more values. Iterables are things that can be iterated over. In this course, we will focus on three types of iterables:

- ranges - iterate over each *value in the range*
- strings - iterate over each *character in the string*
- lists and tuples (which will be covered in the future)

In each iteration of the **for** loop, an element is taken from the iterable and set in turn, i.e. the `variable` in the above example. The variable may or may not be used, depending on the purpose of the loop. Some examples should prove useful:

```
for i in range(5):                # iterate over each value in a range
  print(i, end=" ")               # 0 1 2 3 4
for c in "ABCdef":                # iterate over each character in a string
  print(c, end=" ")               # A B C d e f
for item in [123, "456", 789]:    # iterate over each element in a list
  print(item, end="!")            # 123!456!789
```

2.4 break statement

The **break** keyword is used within both **for** and **while** loops. When Python encounters the **break** keyword and executes it, it ends the innermost loop immediately. A **break** statement is the only way to exit an infinite loop.

In the example below, Python runs each iteration of the loop as usual, until it encounters the **break** statement when `i == 2`. Once it does, it ends the **for** loop and continues executing code after the **for** loop.

```
for i in range(5):          # iterate over i = 0, 1, 2, 3, 4
  if i == 2:                # exits loop when i == 2,
    break                   # preventing loop execution with i = 3, 4
  print(i, end=" ")         # prints 0 1
print("exited loop")
```

2.5 `else` statement

An **`else`** block can be added at the end of **`for`** and **`while`** loops to run *only when the loop was fully executed*, i.e. without encountering a **`break`** statement. Said differently, when Python encounters a **`break`** statement, it will *skip over* the **`else`** block. Compare what is displayed with and without the break statement in the code block below.

```
for i in range(5):
  if i == 2:
    break                # exits loop when i == 2
  print(i, end=" ")      # prints 0 1
else:
  print("else block")    # not printed
print("exited loop")     # prints
```

2.6 `continue` statement

The **`continue`** statement is used within both **`for`** and while loops. When Python encounters the **`continue`** keyword and executes it, it ends the *current iteration*, and *skip ahead* to the next iteration of the loop.

In the example below, Python runs each iteration of the loop as usual, until it encounters the **`continue`** statement when `i == 2`. Once it does, it ends that iteration there, moving on to the next iteration (with `i = 3`), *skipping* the `print()` function completely.

```
for i in range(5):
  if i == 2:
    continue             # skips iteration when i == 2
  print(i, end=" ")      # prints 0 1 3 4
else:
  print("else block")    # prints
print("exited loop")     # prints
```

You can combine the use of **`break`**, **`else`**, and **`continue`** as needed.

3 Nested loops

`while` and **`for`** loops can also be *nested* within each other to repeat over multiple variables or iterations. For example, if we had a 3x3 matrix and wanted to compute its sum, we would have to iterate through every entry in the matrix, which is indexed by a *row* and a *column*.

As a simpler example, to compute the sum of all pairs of products between `[1, 2, 3]` and `[5, 6, 7]`, $(1 \times 5 + 1 \times 6 + \cdots + 3 \times 6 + 3 \times 7)$, we can use two nested **`for`** loops to compute the sum.

```
prod_sum = 0
for i in range(1, 4)
  for j in range(5, 8):
    prod_sum += i * j
print(prod_sum)
```