# Chapter 5 - loops

Goals:

- Explain what loops do and what they are used for.
- Trace the execution of for loops and while loops and correctly predict the state of variables in each iteration.
- Write for and while loops to solve computational problems.
- Asses when to use for loops and when to use while loops.
- Write loops that use the Accumulator pattern to keep running sums or products and/or extend lists.

## 5.0 Loops execute commands once for each value in a collection

- Doing repeated cacluations by hand for large data sets is painful. Loops let you set up the calculation once and apply it to as many data points as you like.
- A for loop tells Python to execute some statements once for each value in a list or some other collection, i.e. "for each thing in this group, do these operations".
- A `while` loop runs continuously until some condition is met

## 5.1 A `for` loop is made up of a collection, a loop variable, and a body

- In the example below, the collection `[2, 3, 5]` is what the loop is being run on. In this example, the collection is a Python list, but other objects can also serve as the collection.
- The body, `print(number)`, specifies what to do for each value in the collection
- The loop variable, `number`, is what changes on each iteration of the loop

```
for number in [2,3,5]:      # for loop over the collection [2,3,5]
    print(number)           # body of the for loop
```

- On the first iteration of the loop, the loop ariable `number` takes the value 2 (the first element of the collection.
- On the second iteration, `number` takes the value 3 (second element of the collection) and so on.
- When the last element of the collection is reached, the loop ends

## 5.2 The first line of a `for` loop must end with a colon, and the body must be indented

- The colon at the end of the first line signals the start of a block of statements
- Python uses indentation rather than conventions in other programming languages (such as `{ }` in C, or `end` in Matlab)
- Any consistent indentation is legal, but almost everyone uses 4 spaces
- If you don't indent, and indent the same amount for every line in the body of the loop, you'll get an error message
- If you indent when you're not supposed to, you'll get an error message
- Be careful how your code is indented!!

## 5.3 Loop variables can be called anything

Loop variables:

- are created on demand (they don't have to be previously defined)
- can be called anything, although its a good idea to have the name of the loop variable reflect something about the collection if possible

While the following is a valid for loop, the loop variable "kitten" probably doesn't add to our understanding of what the loop is for, unless the the collection `[4,5,6]` represents a kitten ID #. 🐱

```
for kitten in [4, 5, 6]:
    print(kitten)
```

### ✅ Skill Check 1

Write a for loop that prints each element of the following list, one item at a time: `[2, 8, 18, 32, 50]`.

## 5.4 The body of a loop can contain many statements

- We recommend structuring your code so no loop is more than a few lines (otherwise your code can be hard to follow
- Here's an example where we let the loop variable be `p` that will loop over a list of prime numbers. The `p` reminds us the loop variable is a prime.
- Our program defines a list of primes and then prints a data table showing the prime, its square and its cube. It also prints a header at the top of the table to label each column

### 🔆 Example: for loop over prime numbers

```
primes = [2, 3, 5, 7, 11]      # define a list of prime numbers
print(" prime  square   cube") # print a header labeling each data column
for p in primes:               # for loop, p will loop over elements of primes
    squared = p**2             # calculate square of p
    cubed   = p**3             # calculate cube of p
    print(f"    {p:2d}     {squared:3d}   {cubed:4d}")  # formatted print
```

- Notice we use a formatted print statement to line the numbers up in columns. The `{squared:3d}` formatting tells python to reserve 3 spaces for digits in the squared numbers and `{cubed:4d}` reserves 4 spaces for cubes.

## 5.5 Iterate over a list of non-numeric items

`for` loops can iterate over lists of non-numeric items in addition to numbers. In the following example, we use a `for` loop to iterate over a list of words. We use a formatted print statement to center-justify them.

```
word_list = ['top','quark','gravity','radiation', 'electromagnetic','pion']

for word in word_list:
    print(f"              {word:^15}")
```

### ✅ Skill Check 2

Create a for loop that iterates over the elements of the following list. Use a print statement inside the loop to print each list element.

```
my_list = ['I', 'am', 'Learning', 'Python']
```

## 5.6 use `range()` to iterate over a sequence of numbers

- The `range(start,stop)` function produces a sequence of integers from `start` to `stop-1`
- `range()` does not produce a Python list (which can take up lots of computer memory for long lists), rather, it generates the numbers on demand, one at a time.

```
for n in range(3,8):
    print("n =",n)
```

- You can also use `range(N)` which generates the numbers: 0, 1, 2, ...N-1

```
for n in range(5):
    print("n =",n)
```

- Or, you can specify a non-unit step: `range(start,stop,step)`. For example, if step = 2, the for loop would begin counting by 2s from `start`.

```
for n in range(0,10,2):
    print("n =",n)
```

### 🔆 Example: counting down

You can count backwards by using a negative `step` and letting the `stop` value be larger than the `start` value

```
for count in range(10,0,-1):
    print(count)
```

### ✅ Skill Check 3

- **Without** running the following code, try to predict what it will print.
- Create a Text Cell below to type your answer. Type exactly what you think the code will output.
- Explain your logic in words as to what the code is doing.
- If you want, you can copy the code in a Code cell and run it to see if you are correct, but this is not required.

```
for t in range(5):
    x = 2*t**2
    v = 2*t
    print(f"t = {t}    x = {x:2.0f}   v = {v}")
```

### ✅ Skill Check 4

Write a for loop to count by fives starting from 5 and ending at 50. Your program should print out each value.

## 5.7 Use `enumerate()` to loop over values and the list index

- In a standard `for` loop, only the values corresponding to each element in a list are accessible. The index isn't.
- The index tells us which element of an array the value is stored in, which can be useful for some applications.
- We could use a counter to keep track of the index ourselves, like this:

```
primes = [2, 3, 5, 7, 11]               # define a list of prime numbers
i = 0                                   # initialized a counter to track the index
for p in primes:                        # loop over elements in primes
    print(f"index = {i}   prime = {p}") # print index and value
    i = i + 1                           # increment the counter
```

- Python provides a way of automatically tracking the index, without having to set up a counter manually
- We use the `enumerate()` function, which returns **two** values: the index and the value for each element of the array
- We "capture" the index returned by `enumerate()` by using a comma
- Here's an example:

```
primes = [2, 3, 5, 7, 11]              # define a list of prime numbers
for i,p in enumerate(primes):          # for loop, i = array imdex
    print(f"index = {i}  prime = {p}")   # print index and value
```

### 🔆 Example

Here's an example where knowing the list index could be helpful:

- Let's create a function that checks to see which elements in a list are integers
- This function uses an `if` statement, which will be discussed in Chapter 6, so don't worry if if this part is confusing. We'll cover it soon.
- Run the code. The result should be a list of list indexes corresponding to the elements that are integers.

```
A = [2,5,8,3.14,'a','b']       # define list containing multiple data types
```

```
int_type = []                # int_type will be a list containing indexes of
                             # intergers in the array. Initialize to empty list

for i,a in enumerate(A):     # loop over elements in list A
    if type(a) is int:       # check to see if the element is an integer
        int_type.append(i)   # if it is, add the index to our list
print("list of indices containging integer data:  ",int_type)
```

### ✅ Skill Check 5

Use the `enumerate()` command to iterate over the following list of items. Print the index and the array element.

```
my_list = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

## 5.8 Use zip() to loop over multiple lists

- The `zip()` function returns the elements for multiple lists
- The lists must be the same length

### 🔆 Example

- In this example we have two arrays, one containing the mass and the other the velocity of a system of particles
- We want to calculate the kinetic energy for each particle in the lists
- We use the `zip()` function to loop over both the mass and velocity in a single loop

```
mass = [0.1, 0.5, 0.9, 2.3]     # masses
vel  = [2.3, 1.2, 3.6, 5.5]     # velocity

KE = []                         # set kinetic energy list to empty list
for m,v in zip(mass, vel):      # loop over both mass and velocity lists
    KE.append(0.5*m*v**2)       # calculate and add KE to KE list

    print(f"m = {m}   v = {v}   KE = {KE[-1]:5.2f}")
```

### ✅ Skill Check 6

- In the following code cell, three lists are defined that count by 2's, 3's and 4's
- Use the `zip()` function to loop over all three lists. Hint: the `zip()` function can handle any number of lists.
- Print three columns of numbers, one for each list.
- Use formatted printing to get the numbers to line up

```
twos   = [2,4, 6, 8,10]
threes = [3,6, 9,12,15]
fours  = [4,8,12,16,20]
```

## 5.9 The accumulator pattern sums or combines elements in a collection of numbers

**Coding patterns** are commonly used blocks of code that serve a particular function. The first coding pattern we will explore is called the **Accumulator Pattern**. It works as follows:

- The Accumulator Pattern consists of a loop and an accumulator variable.
- On each iteration of the loop, the accumulator variable "accumulates" or "gathers" information from the loop

### 🔆 Example: Summing integers

The following example used the Accumulator Pattern to sum the integers from 1 to 10.

- We first initialize the accumulator variable, called `total` to zero.
- We then loop over the the integers from 1 to 10
- On each itereation of the loop, we add the loop variable (our integer) to the accumulator variable (the running total)

```
N = 10                  # N = upper limit of sum

total = 0               # total = accumulator = sum of integers
for i in range(1,N+1):  # loop over integers 1 to N
    total = total + i   # add i to the running total

print("sum of integers from 1 to",N,"is",total)
```

### 🔆 Example: Extending a list

Another example of the accumulator pattern is "accumulating" items in a list.

- Here's an example where we keep appending items on the end of a list based on the prior elements.
- In this example, the new item equals the previous item times two.
- If $x_n$ is the $n^{th}$ list item, $x_{n+1} = 2 \cdot x_n$
- In other words, the $n^{th}$ item in the list will be $x_n = 2^n$.
- Remember, `my_list[-1]` is the last item in the list, so `2*my_list[-1]` multiplies the last element by 2.

```
my_list = [1]                    # initialize the list with 1

for n in range(10):              # for loop will calculate the next 10 items
    my_list.append(my_list[-1]*2)  # next item = previous item * 2

print(my_list)
```

### ✅ Skill Check 7

- Write a program that uses the Accumulator Pattern to calculate the factorial $x!$ of a given whole number `x` where $x! = 1 \cdot 2 \cdot 3 \cdots (x-1) \cdot x$
- Hint: In this application, you will be accumulating the **product** of a series of numbers instead of the **sum**.
- When we calculated the sum, we initialized the accumulator to 0. What should you initialize the accumulator to be when calculating a product?
- Test out your code for N = 6

### ✅ Skill Check 8

- Write a program that calculates the sum $\Sigma_{n=0}^{N} \frac{1}{2^n}$, where the upper limit is stored in a variable `N`
- Test out your code for N = 10

### ✅ Skill Check 9

Write some code to produce the first 15 terms of the Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8...

- The Fibonacci sequence is defined as $x_n = x_{n-1} + x_{n-2}$, i.e. each term is the sum of the previous two terms.
- Start with a list containing the first two terms in the series: `[0,1]`
- Use the accumulator pattern and the example above of extending a list
- Print your list when you are done

## 5.10 Nested loops

- Loops can contain other loops. Loops within loops are called nested loops
- Let's start with a simple loop. The loop variable is `b` and it will loop over the collection `[1, 2, 3]`.
- We'll print out the value of `b` on each iteration of the loop
- Since this is going to be the inner loop of our nested loop, we'll label it "inner loop"
- Finally, we'll print a statement telling us when the inner loop is done.

Here's the code:

```
    print("  inner loop")

    for b in [1, 2, 3]:
        print(f"     b = {b}")
    print("inner loop done")
```

- Now, let's place this loop inside an outer loop.
- The outer loop variable will be `a` that will loop over the collection `[10, 20, 30]`
- We'll print both the values `a` and `b`
- We'll also place the "inner loop done" print statement inside our outer loop (but outside the inner loop)

Here's the code:

```
    print("outer loop    inner loop")  # header info before loops start

    for a in [10, 20, 30]:             # outer loop, a = loop variable

        for b in [1, 2, 3]:            # inner loop, b = loop variable
            print(f"  a = {a}       b = {b}")    # only this statement is performed inside both loops
        print("inner loop done")       # display message after inner loop finishes

    print("outer loop done")           # display message after outer loop finishes
```

- Notice what happens. While the inner loop is looping, the a variable updates while the b variable stays constant.
- When the inner loop is complete, the "inner loop done" statement is printed and the next iteration of the outer loop starts.
- Notice the indentation. Everything with a single tab is executed inside the outer loop. Everything with a double tab (i.e. the inner print statement) is executed inside both the inner and outer loops
- Study this code to make sure you understand it

### ✅ Skill Check 10

Trace the following code by hand (i.e. don't run it). Create a text box and write down what the code will output. If you like, you can copy the code in to a Code Cell to see if your prediction was correct.

```
for x in [1,10,100]:
    s = x
    for y in [0,1,2]:
        s = s + y
        print(x,y,s)
```

## 🔆 Example

The following program prints the numbers 1 to 100 in 10 columns using nested loops

- the inner loop iterates over each column
- the outer loop iterates over each row
- when printing along a given row, we don't want the print statement to return to the next line (i.e. we want to stay on the same line for the next number). We accomplish this by using the `end=""` option in the print statement. We then print a new line using `print()` after the inner loop is done.
- experiment with this code to make sure you understand it.

```
    for row in range(0,100,10):          # loop over rows
        for col in range(1,11):          # loop over columns
            product = row + col          # calculate the sum of the row and col variables
            print(f"{product:3d}  ",end="")   # print result suppressing line feed (don't start new line)
        print()                                # start a new line after previous row is done printing
```

### ✅ Skill Check 11

Create a multiplication table so that the table entry equals the product of the row number and the column number.

- The table should have all product up to NxN, where N is a variable.

- Use the previous example as a starting point.
- As in the previous example, make sure the numbers in the multipication table are lined up in neat columns.
- Run your code with N = 12

### ✅ Skill Check 12

- Modify the program you wrote for Skill Check 8 in the following way:
- Display the sum found in Skill Check 1 for the following upper limits `N = 1, 2, 3, 5, 10, 100`.
- Use a nested loop to Write a program that calculates the sum $\Sigma_{n=0}^{N} \frac{1}{2^n}$.
- The output should be produced by a single print statement inside the outer loop

## 5.11 `while` loops

- A `while` loop iterates while some condition remains true.
- It starts with `while`, specifices a condition to keep looping and ends with a colon `:`

### 🔆 Example

Here's a simple example to print all powers of 2 less than 100

- Note: we want to use a while loop instead of a for loop since we don't know ahead of time how many terms to include (we could do some math to figure it out, but that's too much work)
- We know $2^0 = 1$, so we'll initialize our first power as 1
- Inside our loop, we'll keep multiplying the previous value by 2 to get the next higher power
- Notice we must do the multiplication **after** we print the power. Why is this? Try moving the multiplication before the print statement to see what happens.

```
p = 1            # initialize power
while p < 100:   # loop while power is less than 100
    print(p)     # display current value
    p = p * 2    # multiply power by 2
```

### 🔆 Example: Rolling dice

This example simulates rolling a 6-sided die. We count how many rolls it takes to get a predetermined roll (in this case a 6).

- We use the `random` library to generate random numbers
- The command `random.randint(a,b)` generates a new random integer each time the command is called. The integer is drawn from a uniform distribution between `a` and `b`. We will use `random.randint(1,6)` to generate numbers between 1 and 6 inclusive.
- We define a variable `target_roll` to be the value we want to roll (in our case 6)
- The variable `count` is a counter to count how many rolls it takes to get our target
- `roll` is the result of our random number generator used to simulate the dice roll.
- Our `while` loop keeps iterating as long as the dice roll is not equal to the target.
- Inside the loop, we use `random.randint(1,6)` to simulate our dice roll
- We increment our counter everytime we generate a random dice roll
- If the roll produces the `target_roll`, the condition `roll != target_roll` will be `False` and teh while loop will stop.
- We print the total number of rolls at the end of the program Run the program to see what it produces

```
import random           # import random library to use random number generator

target_roll = 6         # roll that you want to get
count = 0               # counter variable to count number of rolls
roll = -1               # initialize roll to any value other than the target

while roll != target_roll:         # loop while desired roll hasn't
    roll = random.randint(1,6)     # use random number generator to draw number between 1 and 6
    count = count + 1              # increase counter tracking number of rolls
    print("roll = ",roll)          # display current roll

# print message giving total number of rolls
print("Number of rolls to get",target_roll,"=",count)
```

### ✅ Skill Check 13

Modify the Rolling Dice program in the following way:

- Adapt it for flipping a coin (2 possible outcomes instead of 6)
- You might imagine 1 = heads and 2 = tails
- Instead of tyring to get a 6, try to get "heads", i.e. a 1.
- Run it a few times. What's the largest number of coin flips you get **without** getting heads (in other words, getting tails every time)?

## Key Points

- for loops execute commands once for each value in a collection
- `while` loops keep looping as long as a specified condition is true
- A for loop is made of a collection, a loop variable and a body
- The first line of a for loop must end with a colon and the body must be indented
- Loop variables can be called anything
- The body of a loop can contain many statements
- Use range() to iterate over a sequence of numbers
- Use enumerate() to loop over values and the list index
- Use zip() to loop over multiple lists
- The accumulator pattern is a way of summing or combining elements in a collection of numbers
- Nested loops place one loop inside another