Contents

1 Getting started with python programming

Objectives:

- Define and implement Python functions, covering parameters, return values, and proper structure, with an emphasis on variable scope.
- Demonstrate mastery in Python list operations, including indexing, slicing, appending, and sorting.
- Apply both for and while loops effectively in Python, incorporating loop control statements.
- Solve real-world problems by integrating functions and lists into modular and reusable code.

Deliverables

- Provide a Python program demonstrating effective use of list operations, with a brief write-up explaining the rationale.
- Share a Python program showcasing the use of both for and while loops, with an explanation of their roles.
- Submit a Python program solving a real-world problem with functions and lists, accompanied by a brief report.

1.1 Hello, Python

Welcome to Python programming! Having delved into C++ last semester, you're now stepping into the world of Python. You'll find Python to be a versatile and expressive language. In the upcoming experiments, you'll explore Python's simplicity and power, unlocking its potential for rapid development and diverse applications. Get ready for a seamless transition and a deeper understanding of Python's unique features!

Take a look at the code below and guess what it does. If you have no idea, that's fine, see the output of the program next.

```
spam_amount = 0
print(spam_amount)

# Ordering Spam, egg, Spam, Spam, bacon and Spam (4 more
   ↪ servings of Spam)
spam_amount = spam_amount + 4

if spam_amount > 0:
```

```
    print("But I don't want ANY spam!")

viking_song = "Spam " * spam_amount
print(viking_song)
```

```
0
But I don't want ANY spam!
Spam Spam Spam Spam
```

This fun program highlights key aspects of Python code. Let's review it together, step by step.

```
spam_amount = 0
```

Variable assignment: will create a variable called *spam_amount* and assign it the value of 0 using =, which is called the assignment operator.

```
print(spam_amount)
```

Function calls: *print* is a Python function that displays the value passed to it on the screen. We call functions by putting parentheses after their name, and putting the inputs (or arguments) to the function in those parentheses.

```
# Ordering Spam, egg, Spam, Spam, bacon and Spam (4 more
   ↪ servings of Spam)
spam_amount = spam_amount + 4
```

The first line above is a **comment**. In Python, comments begin with the # symbol.

Next we see an example of reassignment. Reassigning the value of an existing variable looks just the same as creating a variable - it still uses the = assignment operator.

```
if spam_amount > 0:
    print("But I don't want ANY spam!")

viking_song = "Spam " * spam_amount
print(viking_song)
```

The first line above is **one of the conditional statements**, often referred to as *if-then statements*, let you control what pieces of code are run based on the value of some Boolean condition.
You can probably guess what this does. Python is prized for its readability and the simplicity.

Note how we indicated which code belongs to the **if**. *"But I don't want ANY spam!"* is only supposed to be printed if spam_amount is positive. But the later code (like *print(viking_song)*) should be executed no matter what. How do we (and Python) know that?

The **colon (:)** at the end of the if line indicates that a new code block is starting. Subsequent lines which are indented are part of that code block.

The later lines dealing with *viking_song* are not indented with an extra 4 spaces, so they're not a part of the if's code block. We'll see more examples of indented code blocks later when we define functions and using loops.

The * operator can be used to multiply two numbers (3 * 3 evaluates to 9), but we can also multiply a string by a number, to get a version that's been repeated that many times.

1.1.1 Numbers in Python

We've already seen an example of a variable containing a number above

```
spam_amount = 0
```

"Number" is a fine informal name for the kind of thing, but if we wanted to be more technical, we could ask Python how it would describe the type of thing that spam_amount is:

```
type(spam_amount)
```

output: *int*

It's an int - short for integer. There's another sort of number we commonly encounter in Python:

```
type(spam_amount)
```

output: *float*

1.1.2 Builtin functions for working with numbers

min and **max** return the minimum and maximum of their arguments, respectively...

```
print(min(1, 2, 3))
print(max(1, 2, 3))
```

```
output:
  1
  3
```

abs returns the absolute value of an argument:

```
print(abs(32))
print(abs(-32))
```

```
output:
  32
  32
```

1.2 Functions and Getting Help

You've already seen and used functions such as *print* and *abs*. But Python has many more functions, and defining your own functions is a big part of Python programming. In this section, you will learn more about using and defining functions.

1.2.1 Getting Help

You saw the abs function in the previous tutorial, but what if you've forgotten what it does?
The *help*() function is possibly the most important Python function you can learn. If you can remember how to use *help*(), you hold the key to understanding most other functions. Here is an example:

```
help(round)
```

Help on built-in function round in module builtins:

```
round(number, ndigits=None)
    Round a number to a given precision in decimal digits.
    The return value is an integer if ndigits is omitted or
       ↪ None.  Otherwise, the return value has the same type
       ↪  as the number.  ndigits may be negative.
```

help() displays two things:

- the header of that function *round*(*number*,*ndigits* = *None*). In this case, this tells us that *round*() takes an argument we can describe as a number. Additionally, we can optionally give a separate argument which could be described as ndigits.
- A brief English description of what the function does.

Common pitfall when you're looking up a function, remember to pass in the name of the function itself, and not the result of calling that function.
What happens if we invoke help on a call to the function round()? Unhide the output of the cell below to see.

```
help(round(-2.01))
```

Python evaluates an expression like this from the inside out. First it calculates the value of *round*(−2.01), then it provides help on the output of that expression. (And it turns out to have a lot to say about integers! After we talk later about objects, methods, and attributes in Python, the help output above will make more sense.) *round* is a very simple function with a short docstring. *help* shine even more when dealing with more complex, configurable functions like *print*. Don't worry if the following output looks inscrutable... for now, just see if you can pick anything new out from this help.

```
help(print)
```

```
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flu
sh=False)

    Prints the values to a stream, or to sys.stdout by defaul
t.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the curre
nt sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a ne
wline.
    flush: whether to forcibly flush the stream.
```

If you were looking for it, you might learn that print can take an argument called *sep* and that this describes what we put between all the other arguments when we print them.

1.2.2 Defining functions

Builtin functions are great, but we can only get so far with them before we need to start defining our own functions. Below is a simple example.

```
def least_difference(a, b, c):
    diff1 = abs(a - b)
    diff2 = abs(b - c)
    diff3 = abs(a - c)
    return min(diff1, diff2, diff3)
```

This creates a function called least_difference, which takes three arguments, a, b, and c. Functions start with a header introduced by the def keyword. The indented block of code following the : is run when the function is called.
return is another keyword uniquely associated with functions. When Python encounters a return statement, it exits the function immediately and passes the value on the right-hand side to the calling context. Is it clear what *least_difference()* does from the source code? If we're not sure, we can always try it out on a few examples:

```
print(
    least_difference(1, 10, 100),
    least_difference(1, 10, 10),
    least_difference(5, 6, 7), # Python allows trailing commas
       ↪  in argument lists. How nice is that?
)
```

Or maybe the *help()* function can tell us something about it.

```
help(least_difference)
```

1.2.3 Functions that don't return

What would happen if we didn't include the *return* keyword in our function?

```
def least_difference(a, b, c):
    """Return the smallest difference between any two numbers
    among a, b and c.
    """
    diff1 = abs(a - b)
    diff2 = abs(b - c)
    diff3 = abs(a - c)
    min(diff1, diff2, diff3)

    print(
        least_difference(1, 10, 100),
        least_difference(1, 10, 10),
        least_difference(5, 6, 7),
    )
```

Python allows us to define such functions. The result of calling them is the special value None. (This is similar to the concept of "null" in other languages.) Without a return statement, *least_difference* is completely pointless, but a function with side effects may do something useful without returning anything. We've already seen two examples of this: *print()* and *help()* don't return anything. We only call them for their side effects (putting some text on the screen). Other examples of useful side effects include writing to a file or modifying an input.

```
mystery = print()
print(mystery)
```

1.2.4 Default Arguments

When we called *help(print)*, we saw that the *print* function has several optional arguments. For example, we can specify a value for *sep* to put some special string in between our printed arguments:

```
print(1, 2, 3, sep=' < ')
```

But if we don't specify a value, *sep* is treated as having a default value of ' ' (a single space).

```
print(1, 2, 3)
```

Adding optional arguments with default values to the functions we define turns out to be pretty easy:

```
def greet(who="Colin"):
    print("Hello,", who)

greet()
greet(who="Kaggle")
# (In this case, we don't need to specify the name of the
   ↪ argument because it's unambiguous.)
greet("world")
```

```
Hello, Colin
Hello, Kaggle
Hello, world
```

1.2.5 Functions Applied to Functions

Here's something that's powerful, though it can feel very abstract at first. You can supply functions as arguments to other functions. Some examples may make this clearer:

```
def mult_by_five(x):
    return 5 * x

def call(fn, arg):
    """Call fn on arg"""
    return fn(arg)

def squared_call(fn, arg):
```

```
    """Call fn on the result of calling fn on arg"""
    return fn(fn(arg))

print(
    call(mult_by_five, 1),
    squared_call(mult_by_five, 1),
    sep='\n', # '\n' is the newline character - it starts a
       ↪ new line
)
```

Functions that operate on other functions are called "higher-order functions." You probably won't write your own for a little while. But there are higher-order functions built into Python that you might find useful to call. Here's an interesting example using the *max* function.
By default, *max* returns the largest of its arguments. But if we pass in a function using the optional *key* argument, it returns the argument x that maximizes *key(x)* (aka the 'argmax').

```
def mod_5(x):
    """Return the remainder of x after dividing by 5"""
    return x % 5

print(
    'Which number is biggest?',
    max(100, 51, 14),
    'Which number is the biggest modulo 5?',
    max(100, 51, 14, key=mod_5),
    sep='\n',
)
```

```
Which number is biggest?
100
Which number is the biggest modulo 5?
14
```

1.3 List

Lists in Python represent ordered sequences of values. Here is an example of how to create them:

```
primes = [2, 3, 5, 7]
```

We can put other types of things in lists:

```
planets = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', '
   ↪ Saturn', 'Uranus', 'Neptune']
```

We can even make a list of lists:

```
hands = [
    ['J', 'Q', 'K'],
    ['2', '2', '2'],
    ['6', 'A', 'K'], # (Comma after the last element is
       ↪ optional)
]
# (I could also have written this on one line, but it can get
   ↪ hard to read)
hands = [['J', 'Q', 'K'], ['2', '2', '2'], ['6', 'A', 'K']]
```

A list can contain a mix of different types of variables:

```
my_favourite_things = [32, 'raindrops on roses', help]
# (Yes, Python's help function is *definitely* one of my
   ↪ favourite things)
```

1.3.1 Indexing

You can access individual list elements with square brackets. Which planet is closest to the sun? Python uses zero-based indexing, so the first element has an index 0.

```
planets[0]
```

Which planet is furthest from the sun?
Elements at the end of the list can be accessed with negative numbers, starting from -1:

```
planets[-1]
```

1.3.2 Slicing

What are the first three planets? We can answer this question using slicing:

```
planets[0:3]
```

planets[0:3] is our way of asking for the elements of planets starting from index 0 and continuing up to but not including index 3. The starting and ending indices are both optional. If I leave out the start index, it's assumed to be 0. So I could rewrite the expression above as:

```
planets[:3]
```

If I leave out the end index, it's assumed to be the length of the list.

```
planets[3:]
```

i.e. the expression above means "give me all the planets from index 3 onward". We can also use negative indices when slicing:

```
# All the planets except the first and last
planets[1:-1]
```

1.3.3 Changing lists

Lists are "mutable", meaning they can be modified "in place". One way to modify a list is to assign to an index or slice expression. For example, let's say we want to rename Mars:

```
planets[3] = 'Malacandra'
planets
```

```
['Mercury',
 'Venus',
 'Earth',
 'Malacandra',
 'Jupiter',
 'Saturn',
 'Uranus',
 'Neptune']
```

That's quite a mouthful. Let's compensate by shortening the names of the first 3 planets.

```
planets[:3] = ['Mur', 'Vee', 'Ur']
print(planets)
# That was silly. Let's give them back their old names
planets[:4] = ['Mercury', 'Venus', 'Earth', 'Mars',]
```

```
['Mur', 'Vee', 'Ur', 'Malacandra', 'Jupiter', 'Saturn', 'Uran
us', 'Neptune']
```

1.3.4 List functions

Python has several useful functions for working with lists. *len* gives the length of a list:

```
# How many planets are there?
len(planets)
```

sorted returns a sorted version of a list:

```
# The planets sorted in alphabetical order
sorted(planets)
```

```
['Earth', 'Jupiter', 'Mars', 'Mercury', 'Neptune', 'Saturn',
'Uranus', 'Venus']
```

sum does what you might expect:

```
primes = [2, 3, 5, 7]
sum(primes)
```

We've previously used the min and max to get the minimum or maximum of several arguments. But we can also pass in a single list argument.

```
max(primes)
```

1.4 Loops

Loops are a way to repeatedly execute some code. Here's an example:

```
planets = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', '
    ↪ Saturn', 'Uranus', 'Neptune']
for planet in planets:
    print(planet, end=' ') # print all on same line
```

The for loop specifies

- the variable name to use (in this case, planet)
- the set of values to loop over (in this case, planets)

1.4.1 range()

range() is a function that returns a sequence of numbers. It turns out to be very useful for writing loops. For example, if we want to repeat some action 5 times:

```
for i in range(5):
    print("Doing important work. i =", i)
```

```
Doing important work. i = 0
Doing important work. i = 1
Doing important work. i = 2
Doing important work. i = 3
Doing important work. i = 4
```

1.4.2 while loops

The other type of loop in Python is a while loop, which iterates until some condition is met:

```
i = 0
while i < 10:
    print(i, end=' ')
    i += 1 # increase the value of i by 1
```

1.4.3 Reading Numbers from List and Printing Tables

This Python code generates and prints multiplication tables for a predefined list of numbers. The outer loop iterates through each number in the list, initiating an inner while loop that computes and displays the multiplication table for that specific number up to 10. The resulting output showcases neatly formatted tables for each number, revealing the products of the number multiplied by integers from 1 to 10. The inclusion of a blank line between tables enhances readability. This simple program demonstrates the use of loops in Python for educational purposes, offering a clear illustration of nested loops and basic arithmetic operations.

```
# List of numbers
numbers = [2, 5, 8]

# Outer loop to iterate through each number
for num in numbers:
    print(f"Table for {num}:")

    # Inner loop (while loop) to print the table
    multiplier = 1
    while multiplier <= 10:
        result = num * multiplier
        print(f"{num} x {multiplier} = {result}")
        multiplier += 1

    print()  # Add a blank line between tables
```

```
Output:
Table for 2:
  2 x 1 = 2
  2 x 2 = 4
  2 x 3 = 6
  2 x 4 = 8
  2 x 5 = 10
  2 x 6 = 12
  2 x 7 = 14
  2 x 8 = 16
  2 x 9 = 18
  2 x 10 = 20

Table for 5:
  5 x 1 = 5
  5 x 2 = 10
  ...
  .
  .
  .
```