ENGG1811: Computing For Engineers

2025 Term 2

Functions, Lists and For-loops

Week 4: Monday 23rd June, 2025

Monday 13:00 - 15:00 | TETB LG34

Today

- Functions
- Lists
- For-loops
- Lab Tips

Reminders

- Assignment 1
  - Due 5pm, Thursday 17th July (Week 7)
  - Start it early!
  - Help sessions are available - but go early in the term as they often get busy closer to the assignment due date!
- Live coding sessions are on Fridays from 1–2 PM
- PASS session are still on

Functions

Functions: Introduction

- We should all already be intimately familiar with functions in one form or another
  - Examples of functions we have used in python are: `math.cos()`, `math.sin()`, `print()`
  - Examples of functions we are probably aware of from maths: $f(x) = x^2, f(x) = \ln(x), f(x, y) = \sqrt{x^2 + y^2}$
- The goal of this lab is to get everyone comfortable creating their own functions
  - Allows us to reuse logic without having to copy-and-paste code
  - Can abstract away details to make our code more readable

Functions: Introduction

- **Question:** What are the key details we need to specify in order to define a function?
  - Inputs
  - Outputs
  - Name
  - Rule/operation
- For $f(x, y) = \sqrt{x^2 + y^2} = z$ what are the:
  - Inputs?
  - Output?
  - Name?
  - Rule/operation?

Functions: Structure

- All functions have the same basic structure
  - Heading : which defines the inputs and the name of the function
  - Body : which specifies our rule/operation
  - Return : which specifies the output
- Sample:

```
def function(input1, input2, ...):    #Heading
    Some sort of code                 #Body
    return output1, output2, ...      #Return
```

- **Questions**
  - What do we think `def` is telling us?
  - What do we think `return` is telling us?
  - Can we specify a function with **no inputs**? How about one with **no outputs**?

Functions: Examples

- Suppose that the only code in a file was this function, and nothing else unless specified:

```
def squarepluscube(num):
    square = num ** 2
    cube = num ** 3
    answer = square + cube
    return answer
```

- **Questions:**
  - What is the name of the function? What is the input? What is the output? What is the rule/operation?
  - What is the result of `squarepluscube(2)` ?
  - What would be the result of `print(square)` , `print(cube)` , or `print(answer)` ?

Functions: Examples

- Suppose that the only code in a file was this function, and nothing else unless specified:

```
def multiplier(x, y, z):
    answer = x * y + z
    return answer
```

- **Questions:**
  - Let `x = 1, y = 1, z = 1`
    - what is `multiplier(x, y, z)` ?
    - what is `multiplier(x, z, y)` ?
  - Let `x = 2, y = 6, z = 3`
    - what is `multiplier(y, z, x)` ?
    - what is `multiplier(z, z, z)` ?
  - Let `v = 2, a = 0, z = 9`
    - what is `multiplier(v, a, z)` ?
    - what is `multiplier(z, v, a)` ?

Functions: Examples Take Away

Take Away

The names of the variables you place as inputs do not matter!

Rather, it is the relative positioning of the inputs that matters.

- In the `multiplier` example, the variable:
  - ' `x` ' is a *parameter* used to represent the **first input**.
  - ' `y` ' is a *parameter* used to represent the **second input**.
  - ' `z` ' is a *parameter* used to represent the **third input**.

Remark

We call the placeholders/symbolic values used to define a function parameters and the actual values provided for a given function arguments.

Lists

Lists: Introduction

- **Lists are containers for data!**
  - The container *inherently* has an order which is referred to as an *index*
    - There is a first element, as well as a second, and so on ...
  - You are allowed to change what is in the container
    - You can change a value of the container at a specific location
    - You can add and remove values from the container
- **Questions:**
  - Do we think that this is a useful concept?
  - When should we *not* use a list?

Defining Lists

- There are a few ways to create a list
  - Direct specification:
    - `list1 = [1, 2, 3]`
    - `list2 = [10, 100, 1000, 10000, 100000]`
  - Arithmetic operations:
    - `list3 = [6] * 3`
    - `list4 = [6] + [7] + [8]`
    - **Question:** How about '−' and '/'?
  - Appending:
    - `list5 = []`
    - `list5.append(9)`
- There is no restriction on what's in the container!
  - `list6 = ["string", 8, 9.5, [1, 2, 3], -6]`
  - **Questions:** Although we *can* do this, does that mean we *should*? What self-imposed restrictions should we place on ourselves for our peace of mind?

For-loops

For-loops: Introduction

- Loops are by *far*, the most important concept of programming
- To understand why it's so important, we need to remember that the whole *point* of programming is to automate tasks that we don't want to do!
  - Computers are very good at doing the *same* task, *repeatedly*
  - If someone told you to square the first 1000 positive integers, by hand, how long do you think it would take you?
    - I'm sure all of us would get tired of the process before we even made a dent towards completion
    - Computers, however, do not get tired, nor do they get bored! So they happily chow down this task in a fraction of a millisecond

For-loops: Introduction

- Let's take our example of squaring the first 1000 positive integers.
- What are the important considerations of this problem?
  - We have some raw value that has not yet been 'processed'
  - There's a set process/procedure that we wish to do to *this value*
  - There's a list/range of values that we wish to repeat our procedure over
- "For every integer value between 1 and 1000, I want to square the value and then immediately print the result"

For-loops: Structure

- All for loops have the same basic structure
  - Heading : which defines the *symbolic name* of each value, and the *list* we are *iterating over*
  - Body : specifies the rule/procedure we wish to repeat
- Sample:

```
for x in some_list:      #Heading
    #Some sort of code   #Body
```

- **Questions:**
  - This is a very similar *structure* to functions: why don't we need some kind of *return statement* here?
  - Is the name `x` important? Could it have been changed? Can I use `x` outside the scope of the loop? Do the answers change if we instead consider `some_list` ?
  - What is the *only* thing we are not allowed to do in the body of the loop?

For-loops: Examples

- "For every value between 1 and 1000, I want to square the value and then immediately print the result"
  - ```
    for each_value in range(1, 1001):
        squared_value = each_value ** 2
        print(squared_value)
    ```
- "For every **even value** between 0 and 1000, I want to square the value and the immediately print the result"
  - ```
    for each_value in range(0, 1001):
        if (each_value % 2 == 0):
            squared_value = each_value ** 2
            print(squared_value)
    ```
  - ```
    for each_value in range(0, 1001, 2):
        squared_value = each_value ** 2
        print(squared_value)
    ```
    ```

For-loops: Examples (Cont)

- "For every item on my desk, I want to print only the items which have a name that is longer than 5 letters, otherwise I want to print boo!"
  - ```
    desk_items = ["keyboard", "mouse", "book",
                  "glasses", "mug"]
    for item in desk_items:
        if len(item) > 5:
            print(item)
        else:
            print("boo!")
    ```
    ```

For-loops: Using append()

- From earlier, we learned that we can use `append` to add values to the end of a list:

```
list1 = [1, 2, 3, 4]
list1.append(5)

# Prints [1, 2, 3, 4, 5]
print(list1)
```

- We can combine this functionality with for loops to create new lists based on existing lists:

```
list2 = []
for num in list1:
    double = num * 2
    list2.append(double)

# Prints [2, 4, 6, 8, 10]
print(list2)
```

List Comprehension

We can combine for-loops and append (more generally simple for-loops) to create lists in one line. We can simplify this:

```
list2 = []
for num in list1:
    double = num * 2
    list2.append(double)

# Prints [2, 4, 6, 8, 10]
print(list2)
```

To this:

```
list2 = [num * 2 for num in list1]
```

Think about how we structure this statement:

```
new_list = [action(variable) for variable in old_list]
```

Lab Tips

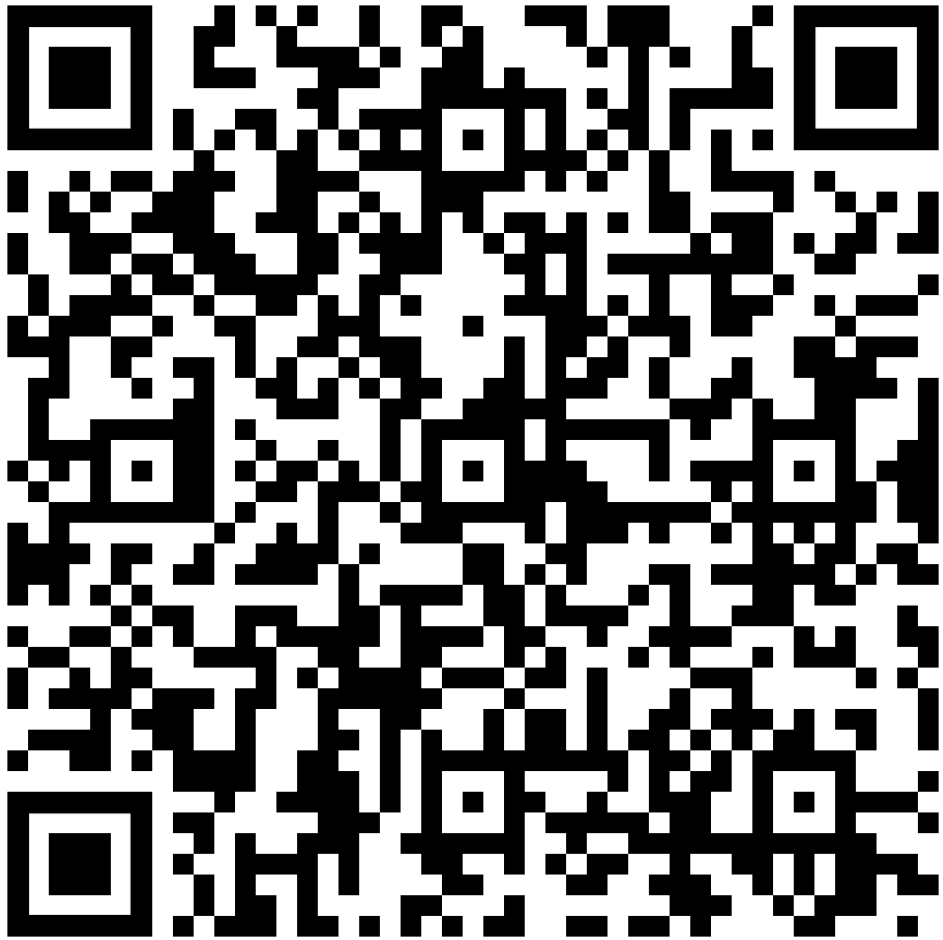
Tips

Tips for the lab today:

- Exercise 1:
  - Remember that functions do not require the input variables to have the same name
  - You need to use the function you created in Task 1 inside the function for Task 2
- Exercise 2:
  - Remember to use `range` and list comprehension in task 1.
  - Task 3: Tasks 1 & 2 are essential for doing task 3 correctly.
- Exercise 3:
  - Making the force list is very tricky! Remember that `range()` cannot do non-integer step sizes
  - Read the hints provided in the exercise carefully, they will be useful

Feedback

Feel free to provide anonymous feedback about the lab!



Feedback Form