

**INF1-IP**

# Introduction to Programming

Lecture 4 – Repetition 1: Loops



# ! Assignment 1 – released today!

You will find the Assignment 1 worksheet available through Learn.

You have until your tutorial on **Friday October 16<sup>th</sup>** to work on it.

You can do this however you like: on your own, with others in the course, with others not in the course, with ChatGPT (see next slide).

We recommend that you write a report with your answers – *but this report is not submitted or assessed.*

You can start on this now – but it covers material taught up to Week 4, so you may not know it all yet.

# ! Assignment 1 – released today!

In your Week 4 tutorial, you will:

- spend the first half of the tutorial discussing the worksheet with your group: what were you not sure about, what did you learn, etc?
- The second half of the tutorial will be the assessment:
  - You will be asked questions *similar to but not identical to* the questions on the worksheet, that should be feasible to work out if you have understood the worksheet.
  - You will be asked a subsection of the questions that will cover key aspects of what we have learnt so far.

# ! Assignment 1 – devices

Assignment are done in tutorials *on your own device*.

Bring along a laptop or something that you can type wordy answers on. If you do not have a laptop, talk to your SA about getting one you can use.

If you turn up without a device, or have problems with your device that means you can't use it:

- You will spend the first half of the tutorial discussing with your peers as normal
- You will leave the tutorial when the assessment begins
- You will go to a session in AT 6:06 later on the same day (time tbc) where you will be able to take the assessment on a DICE machine.

# ! Class tests - devices

Class tests are generally also done on your own device, but they are held in AT 6:06, so anyone without a functioning device can use a DICE machine without changing place, time, etc.



# Rules for in-tutorial assessment and class tests

1. You **must not** have any apps or windows open other than the one containing the assessment. Your screen must be visible at all times.
2. You **must not** have with you phones or any other devices other than the one you are working on – these *must* be in bags or handed in to tutors / invigilators
3. You **must not** communicate with anyone else during the assessment.

Both in-tutorial assessment and class tests are done under exam conditions.

If you are caught breaking any of these rules you will be asked to leave the session and will not be permitted to retake assessment during that assessment point.

# ! Grading

The class tests are *auto marked* and you will know immediately whether you have passed, and if not, which parts you still need to work on.

The assignments are *hand marked* and you will get your grade back approximately 10 days after submission.

Part 1 timetable:

Assignment 1 – Friday of Week 4 – grade received Monday of Week 6  
(before lecture)

Class Test 1 – Thursday of Week 5 – grade received Thursday of Week 5

Part 2 begins Monday of Week 6.

Students either progress to Part 2, or work on developing Part 1.

# > Working with AI

**Less useful for learning:** *"Give me the code to solve this."*

**A useful start:** *"Explain how this code works."*

**Better:** *"Here is my attempt, what I expected, and what happened. Give me a hint to help me find the problem, rather than fixing it for me."*

**Strong practice:** *"Give me a similar problem, one at a time. Wait for my attempt, then critique my code and reasoning. Let me revise it before showing a solution."*

**AI can be wrong. Run the code, test its suggestions, and make sure you can explain your solution yourself.**

# # Loops

# > Why are loops important?

Computers are really good at doing things over and over.

Also, processing things (data, computation, etc) often involves doing things over and over.

So *looping* is a core part of coding.

In Python there are two main kinds of loops:

- `while`
- `for`

# > While loops

While loops have a *condition*.

*While* that condition is true, they keep executing  
If that condition becomes untrue, they stop

```
while condition:  
    do_something
```

# > While loops

Loops must have a *terminal point* - otherwise they run forever.

- Often, loops have a counter that is moving towards the terminal point
- Sometimes there is an external condition that is being checked, and the loop continues until the status of that condition changes in some defined way

## > While loops: counter

```
no_students = 0                # counter must be initialised

while no_students < 10:        # check if the stop condition has been reached
    no_students += 1           # update the counter
    print(f"current enrolment is {no_students}") # do something in while loop

print("course is full")        # (optional) do something when loop terminates
```

## > While loops: external condition

```
while in_semester:                                # check stop condition

    print("I am attending lectures")               # do something in the loop
    in_semester = check_dates()                   # check if the stop condition has been
                                                # updated. Here we are assuming there is
                                                # a function check_dates that has been
                                                # defined elsewhere that returns True if
                                                # in semester and False otherwise

print("I am done with lectures")                  # (optional) do something when loop terminates
```

# { Student notable task

- You are heading out for a night out and are waiting for five friends to arrive at your flat.
- Initially, no one has arrived. Once everyone is there, you will head out
- Create a while loop that will:
  - Print out how many friends are currently there every time a new person arrives (you don't need to a special way to check this – just increase the number of people every time the loop runs)
  - Once five friends have arrived it will terminate and print out a message to say you are going out.

# { Student noteable task 2 – if time

- You are out on your night out and having a lot of fun, but maybe it's time to go home.
- Ask your friends if they want one more tune. If so, print a message to say the party is still going. Keep on doing this until they don't want another.
- If they don't want another tune, print a message to say it's time to go home.

# { Student noteable task – solution\*

```
friends_arrived = 0

print(f"Friends here: {friends_arrived}")

while friends_arrived < 5:
    friends_arrived += 1
    print(f"Friends here: {friends_arrived}")

print("Everyone's here. Time to head out!")
```

\* Other solutions are possible. For example, you don't have to use formatted printing and you could check `friends_arrived <= 4`

# { Student noteable task 2 – solution\*

```
keep_playing = True

while keep_playing:
    answer = input("One more song? (yes/no): ").strip().lower()

    if answer == "yes":
        print("Playing another anthem!")
    else:
        keep_playing = False

print("Music off. Time to head home!")
```

\* Other solutions are possible.

# ! Don't get stuck in an infinite loop

- It is easy to write a while loop that runs forever
- Make sure that the stop condition can be reached
  - Make sure it's being properly checked
  - Make sure the counter is being incremented in the right way (do you want it to count up or down? Or do something else?)

# > For loops

*While* loops keep on running until a stop condition is reached.

*For* loops run for a set number of times

- Working through *a range*
- *Iterating* through a sequence of things such as a list

# > range()

Python has a built-in function for producing a range of integers – *range()*.

There are three variations:

- *range(stop)* – starts at 0 and counts up to **but not including** *stop* (which must be an integer)
- *range(start, stop)* – starts at *start* and counts up to **but not including** *stop* (both must be integers)
- *range(start, stop, step)* – starts at *start* and goes up to **but not including** *stop*, but instead of including every integer, it takes *step* steps at a time (all must be integers).

```
for number in range(..):  
    do_something(number)
```

## > range()

range(stop) and range (start, stop) only count upwards

e.g,

```
for number in range(10, 5):  
    print(number)
```

Does not return anything.

But range(stop, start, step) can count backwards if given a negative step

```
for number in range(10, 5, -1):  
    print(number)
```

Returns 10, 9, 8, 7, 6

# > Iteration

The other way to move through a for loop is through *iterating* through a *collection*.

We'll learn more about collections later, but for now we will look at iterating through a string

```
word = "hello"  
  
for character in word:  
    print(character)
```

# > Accumulators

A common way of using loops is to build up a value over time – e.g., a score, count or piece of text.

This is done using an *accumulator* – a variable that is initially instantiated (often to zero, empty string, etc), and iteratively changed in the for loop.

# > Accumulators

```
no_evens = 0

for number in range(1,10):
    if number % 2 == 0:
        no_evens += 1

print(no_evens)
```

# { Student Noteable task

Write a for loop for a simple word game:

For every letter in a word that the user enters,

- It scores 3 if it's a vowel (or just pick one letter if you're not sure how to do multiple)
- It scores 1 otherwise

Return the total score of the word.

# { Student Noteable task – solution\*

```
word = input("Enter a word to discover its score")
score = 0

for letter in word.lower():
    if letter == "a" or letter == "e" or letter == "i" or letter == "o" or letter == "u":**
        score += 3
    else:
        score += 1

print(f"Your word {score} points!")
```

\*\*there is a more concise way to do this which we will look at in the next lecture

## > Loop control - break

Sometimes we want to jump out of loops under certain conditions

*e.g., is there an 'a' in a given word? We want to work through the word letter by letter until we find an 'a'. But if we do, we don't need to stay in the loop.*

For this, we use *break*.

```
word = "complicated"

for letter in word:
    if letter == "a":
        print("this word has an 'a' in it")
        break
```

## > Loop control - continue

Sometimes we want to stay in a loop but only perform the loop action in certain circumstances. For this, we use *continue*. This tells the compiler to skip ahead to the next iteration.

*e.g., how many 'a's are there in a given word? We want to check if something is an 'a' and add to an accumulator if so, and if not just skip ahead to the next iteration.*

```
word = "complicated"
number = 0

for letter in word:
    if letter != "a":
        continue
    number += 1
print(f"number of a's is {number}")
```

## > Loop control - pass

This does nothing at all – and is useful as a placeholder.

```
for number in range (1,10):  
    if number % 2 == 0:  
        # hmm ... what do I need to do here?  
  
    print(number)
```

This will produce an error when you run it

```
for number in range (1,10):  
    if number % 2 == 0:  
        pass # hmm ... what do I need to do here?  
  
    print(number)
```

This will run – and just ignore the if loop as nothing is actually happening in it. You can go back later and fill it in.

# > Nested loops

Just as we can nest if statements, we can also nest for statements.

*For example:*

*At a pub quiz, you have a number of teams with four players on each.*

*For each team, you want to print out a list of all the players.*

The outer loop loops through the teams, and an inner loop loops through the players.

# { Student Noteable task

Create a program that will list the all the teams in a pub quiz with all their players - teams are called Team 1, etc., and each team consists of Player 1 – Player 4.

# { Student Noteable task – solution\*

```
for team in range(1,11):  
    print(f"Team {team}:")  
  
    for player in range(1,5):  
        print(f"    Player {player}")
```

Output:

Team 1:  
 Player 1  
 Player 2  
 Player 3  
 Player 4  
Team 2:  
 Player 1  
 Player 2  
 Player 3  
 Player 4  
Team 3:  
 Player 1  
 Player 2  
 Player 3  
 Player 4  
Team 4:  
 Player 1  
...

# > Summary

In this lecture, we looked at:

- Repeating code with while and for loops
- Generating sequences of integers with range()
- Building totals and counts with accumulators
- Altering loop flow with break and continue
- Using pass as a placeholder for an unfinished body
- Tracing nested loops