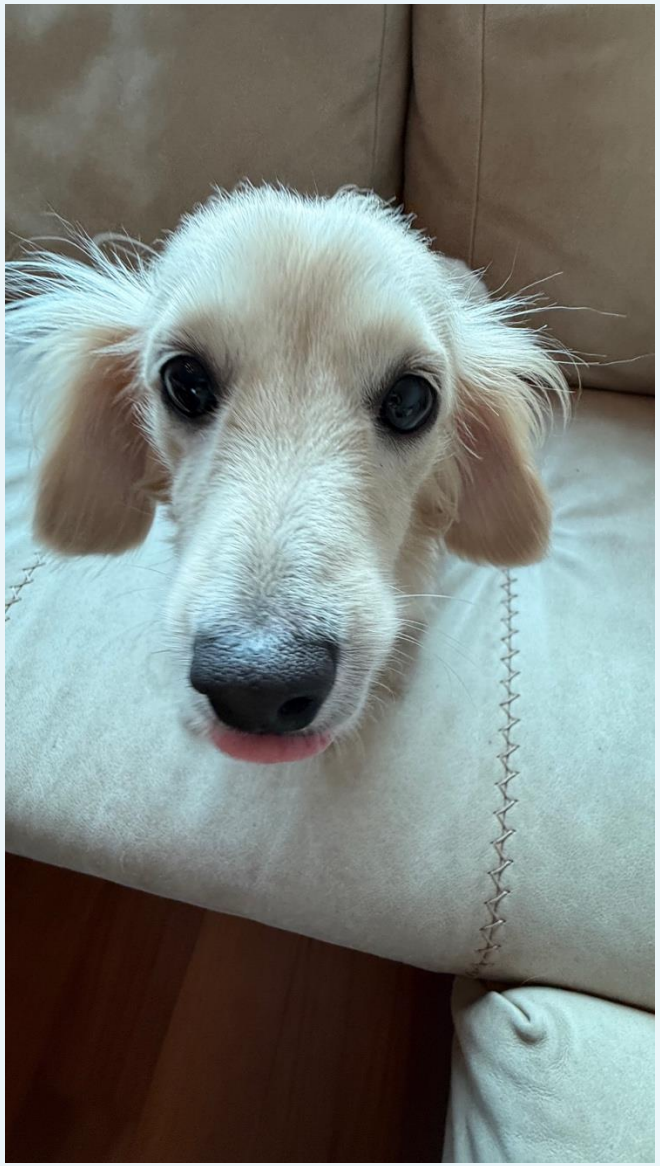
INF1-IP

Introduction to Programming

Lecture 19 – Input/Output 2 – Working with time



Pet of the lecture: Whiskey

Intro to Programming End of Semester Survey 2026



> Reading CSV files

In Lecture 10, you looked at how to read and write text files. Today we will look at CSV files.

- What are CSV files?
- A **C**omma **S**eparated **V**ariable file is a plain-text file used to store data in a table, like a simple spreadsheet
- Each **line** represents a row
- **Commas** separate the values into columns
- The first line often contains column names

```
Name,Age,Course
Alex,19,Computing
Sam,20,Mathematics
```

Let's code that – a csv file for sales in a café

Python has a build-in module to help us with that. We'll also use `Path`.

```
import csv
from pathlib import Path
```

First, we create the file that we want the data to go into.

- Create a variable called `data_dir` and set its path to "`selfguided_output`"
- Use mkdir to make this object into a directory if it doesn't already exist in the right location, and if it does already exist, do nothing.
- Within that directory, create a file called "`cafe_sales.csv`", and set that to a variable called `csv_path`

```
data_dir = Path("selfguided_output")
data_dir.mkdir(exist_ok=True)
csv_path = data_dir / "cafe_sales.csv"
```

Let's code that – a csv file for sales in a café

This variable `csv_path` tells Python where we want it to write to.

Next we create the data for the café. This will be a list of lists: each list is a row we want to create in the csv file.

```
sample_rows = [
    ["item", "timestamp", "price_gbp", "quantity"],
    [" Latte, large ", "2026-02-10T09:00:00+00:00", " 3.50 ", "2"],
    ["Tea", "2026-02-10T09:15:00+00:00", "2.20", ""],
    ["Hot chocolate", "2026-02-10T09:30:00+00:00", "bad", "1"],
    ["Cappuccino", "2026-02-10T09:45:00+00:00", "3.20", "3"],
]
```

Let's code that – a csv file for sales in a café

Next we write that data to our new file.

- We use the `with` method that you met in Lecture 10.
- Then we use a method called `csv.writer(file)` to create an object that writes csv rows.
- Finally, we apply the method `writerows(sample_rows)` to this object so that it writes the rows we have created to our new csv file.

```
with open(csv_path, "w", newline="", encoding="utf-8") as file:
    writer = csv.writer(file)
    writer.writerows(sample_rows)
```

Let's code that – a csv file for sales in a café

```
import csv
from pathlib import Path

data_dir = Path("selfguided_output")
data_dir.mkdir(exist_ok=True)
csv_path = data_dir / "cafe_sales.csv"

sample_rows = [
    ["item", "timestamp", "price_gbp", "quantity"],
    [" Latte, large ", "2026-02-10T09:00:00+00:00", " 3.50 ", "2"],
    ["Tea", "2026-02-10T09:15:00+00:00", "2.20", ""],
    ["Hot chocolate", "2026-02-10T09:30:00+00:00", "bad", "1"],
    ["Cappuccino", "2026-02-10T09:45:00+00:00", "3.20", "3"],
]

with open(csv_path, "w", newline="", encoding="utf-8") as file:
    writer = csv.writer(file)
    writer.writerows(sample_rows)
```

Let's code that – a csv file for sales in a café

We can check that our new directory exists, and that it contains our new csv file, and we can look at what is in the new file.

We can also use

```
print(csv_path.read_text(encoding="utf-8"))
```

to print out what is in the file.

> Working with dates and times

Python has a build in module called `datetime` that provides classes for working with date and time. Two important modules are:

- `date` stores a year, month and day
- `datetime` stores a date *and* a time of day

Datetime refers *both* to the module and a class it contains – so make sure you know which you are talking about. This should be clear from context.

Dates and times are always given with the largest component first.
e.g., the date and starting time for this lecture is:
2026-11-23 16:10:00 → yyyy-mm-dd hh:mm:ss

> Working with dates and times

The `datetime` module allows Python to know what dates are possible

- e.g., trying to create a date of December 32rd or February 30th would create `ValueError`
- February 29th 2026 would also a `ValueError` but February 29th 2028 would not.

N

```
lecture_start = datetime(2026, 2, 19, 9, 0)

print(lecture_start.year, lecture_start.month, lecture_start.day)
print(lecture_start.hour, lecture_start.minute)
```

```
2026 2 19
9 0
```

> Working with dates and times

The `datetime` module allows Python to know what dates are possible

- e.g., trying to create a date of December 32rd or February 30th would create `ValueError`
- February 29th 2026 would also a `ValueError` but February 29th 2028 would not.

Note, don't prefix single digits with a Zero

```
lecture_start = datetime(2026, 02, 19, 09, 0)

print(lecture_start.year, lecture_start.month, lecture_start.day)
print(lecture_start.hour, lecture_start.minute)
```

```
  Cell In[13], line 1
    lecture_start = datetime(2026, 02, 19, 09, 0)
                                    ^
SyntaxError: leading zeros in decimal integer literals are not permitted; use an 0o prefix for octal integers
```

> `Timedelta` objects

`Timedelta` objects are used for representing *how long* things last.

We can use a start point and a time delta to find out when things will end.

e.g., I'm going to keep working until the end of December 23rd, and then I'm going to take two weeks off. When am back at work?

{ Let's code that

```
from datetime import datetime, timedelta

holiday_start = datetime(2026, 12, 24, 9, 0)
holiday_length = timedelta(days=14)
holiday_end = holiday_start + holiday_length
next_celebration = holiday_start + timedelta(weeks=1)

print(holiday_end)
print(next_celebration)
```

```
2027-01-07 09:00:00
2026-12-31 09:00:00
```

> `Timedelta` objects

We can also create a `timedelta` object by adding or subtracting time points.

In the last example, I worked out when I would be back at work using a `timedelta` of 14 days for my holiday. But actually I finish work in the evening and start in the morning, so let's compare actual start and end points, and see what the time period actually is:

```
start = datetime(2026, 12, 23, 17, 0)
end = datetime(2027, 1, 7, 9, 0)
duration = end - start

print(duration)
print(duration.total_seconds() / 3600, "hours")
```

```
14 days, 16:00:00
352.0 hours
```

> Converting between dates and strings

A common way to represent dates is using **ISO 8601.** This creates the format we have seen earlier in the lecture, but with a T separating the date and time:
yyyy-mm-ddThh:mm:ss → 2026-11-23T16:10:00

The `datetime` module provides methods to convert between this format and strings:
`datetime.fromisoformat()` reads a string in this format and converts it to a `datatime` objects.
`.isoformat()` read a `datetime` object and converts it to a string.

> Converting between dates and strings

```
timestamp_text = "2026-11-19T09:00:00"
timestamp = datetime.fromisoformat(timestamp_text)

print(type(timestamp_text))
print(type(timestamp))
print(timestamp + timedelta(minutes=50))
print(timestamp.isoformat())
```

```
<class 'str'>
<class 'datetime.datetime'>
2026-11-19 09:50:00
2026-11-19T09:00:00
```

> Converting between dates and strings

ISO 8601 is good because it is unambigious. But it's not how most of us do dates in every day life. But how we do approach this varies – so if we use other formats we have to be explicit about what we mean.

`datetime.strptime(text, format)` converts `text` to a datetime object using the format specified by `forma`

`Format` tells python how to interpret each element of `text` according to this code:

Code	Meaning	Example
`%Y`	Four-digit year	`2026`
`%m`	Month number	`11`
`%d`	Day of the month	`19`
`%B`	Month name	`November`
`%H`	Hour, using the 24-hour clock	`09`
`%M`	Minute	`00`
`%S`	Second	`00`

> Converting between dates and strings

What does this print:

```
appointment = datetime.strptime("03/04/2026 09:30", "%d/%m/%Y %H:%M")
print(appointment.strftime("%d %B %Y at %H:%M"))
```

```
03 April 2026 at 09:30
```

If we are American, we can use exactly the same approach – we just have to tell Python to do the formatting differently:

```
appointment = datetime.strptime("03/04/2026 09:30", "%m/%d/%Y %H:%M")
print(appointment.strftime("%d %B %Y at %H:%M"))
```

```
04 March 2026 at 09:30
```

> Time zones

So far we have looked at *naïve* datetimes, which do not consider time zones. This is fine if you are always working in the context of a single time zone, but often we have to navigate more than one.

We can use an *offset* to indicate how the time we are talking about compares to UTC (Coordinated Universal Time).

What `datetime` is it Edinburgh now?
2026-11-23T17:30:00 → this is true locally, but if we are using this in a different time zone we might cause errors. This is a *naïve* datetime.

> Time zones

So far we have looked at *naïve* datetimes, which do not consider time zones. This is fine if you are always working in the context of a single time zone, but often we have to navigate more than one.

We can use an *offset* to indicate how the time we are talking about compares to UTC (Coordinated Universal Time).

What datetime is it Edinburgh now?
2026-11-23T17:30:00+00:00 → this is true everywhere in the world, because we are being specific about time zone. This is an *aware* datetime.

> Elapsed time

We can use `perf_counter()`. This gives us a time in seconds: the time that has elapsed since the clock's reference point it.

This in itself is not useful since we don't know the reference point is – e.g., we can't assume it's when we last rebooted.

But we can use it to work out how much time elapses between two events (e.g., how long it takes code to run) by using it at two points and comparing.

> Elapsed time

We can use `perf_counter()`. This gives us a time in seconds: the time that has elapsed since the clock's reference point it.

This in itself is not useful since we don't know the reference point is – e.g., we can't assume it's when we last rebooted.

But we can use it to work out how much time elapses between two events (e.g., how long it takes code to run) by using it at two points and comparing.

```
start = perf_counter()
# Code to time goes here
end = perf_counter()

elapsed = end - start
```

> Elapsed time

We can also use sleep() to get the code to pause for a given number of se

```
from time import perf_counter, sleep

start = perf_counter()
sleep(0.5)  # Pause for half a second
end = perf_counter()

print(f"Elapsed time: {end - start:.3f} seconds")
```

```
Elapsed time: 0.501 seconds
```

The elapsed time may not be exactly the amount of time we asked the code to sleep for – as the computer may be doing other things in the background.

> Asynchronicity

So far, we've looked at doing things one step at a time. But sometimes this makes no sense.

I made supper last night. In a synchronous approach this might look like:

- Put the rice on to boil. Wait for the rice to boil. Drain the rice.
- Chop vegetables and put on to fry. Wait for veg to fry. Remove veg from heat.
- Lay the table. Eat.

A synchronous approach means we are waiting for each activity to complete before we are starting the next one.

> Asynchronicity

So far, we've looked at doing things one step at a time. But sometimes this makes no sense.

I made supper last night. An *asynchronous* approach this might look like:

- Put the rice on to boil.
- Whilst it is boiling, chop the veg and put on to fry
- Whilst these are both cooking, lay the table
- Remove the rice and veg from the heat and serve

An asynchronous approach means there are different activities (cook rice, cook veg, lay table) that are happening at the *same time*.

> Asynchronicity

Asynchronicity is a really important part of efficient coding.

It is a way of implementing *concurrency* – which means you are making multiple progress on different tasks over the same period.

It is different to *parallelism* – which means doing different tasks at **exactly the same time.**

In the cooking analogy, parallelism would be getting someone else to help me. I could cook rice and my husband could make the veg – he would be my parallel processor.

> `async` and `await`

When we are defining functions, we can use `async` to define it as a *coroutine function*: a function that can pause whilst waiting for something.

Imagine we are getting readings from weather stations. We know that there might be a delay, so we define this method as a coroutine function so that it can pause whilst waiting.

`asyncio.sleep()` is similar to `time.sleep()` – except it allows other asynchronous work to make progress during the wait.

We use `await` to tell it how long to sleep for.

{ Let's code that

```
import asyncio

async def wait_for_station(name, delay):
    print("Requesting a reading from", name)
    await asyncio.sleep(delay)
    print("Received a reply from", name)
    return name

result = await wait_for_station("Meadows", 0.3)
print("Result:", result)
```

> Performing actions one after the other

Just adding async doesn't necessarily mean things happen concurrently.

For example, what will happen here?

```
before = time.perf_counter()

first = await wait_for_station("Meadows", 0.3)
second = await wait_for_station("Leith", 0.1)

sequential_elapsed = time.perf_counter() - before
print([first, second])
print(f"Time taken: {sequential_elapsed:.2f} seconds")
```

```
Requesting a reading from Meadows
Received a reply from Meadows
Requesting a reading from Leith
Received a reply from Leith
['Meadows', 'Leith']
Time taken: 0.40 seconds
```

Here, we are telling Python: first, get the Meadows reading, and then get the Leith reading. The time taken is (more or less) the sum of the two actions.

> Performing actions one after the other

But we can run asynchronous coroutines concurrently if we want.

We can use `asyncio.gather()` to group together coroutines we want run at the same time. Getting a reading from the Meadows is not dependent on getting a reading from Leith, so we can do it at the same time.

```
before = time.perf_counter()

station_names = await asyncio.gather(
    wait_for_station("Meadows", 0.3),
    wait_for_station("Leith", 0.1),
)

concurrent_elapsed = time.perf_counter() - before
print(station_names)
print(f"Time taken: {concurrent_elapsed:.2f} seconds")
```

```
Requesting a reading from Meadows
Requesting a reading from Leith
Received a reply from Leith
Received a reply from Meadows
['Meadows', 'Leith']
Time taken: 0.30 seconds
```

The time take is the largest wait, not the sum of the weights.

> JSON data

Java**S**cript **O**bject **N**otation (JSON) is a text format for storing and exchanging structured data.

Despite its name it is language-agnostic and is used by lots of different programming languages. It is common to use it whilst working with a web API.

Python has a module called `json` which turns Python types into JSON types.

> JSON data

Python has a module called json which turns Python types into JSON types.

JSON	Python
Object	`dict`
Array	`list`
String	`str`
Number	`int` or `float`
`true` / `false`	`True` / `False`
`null`	`None`

Note that in Python we can use either single or double quotation marks for strings or object keys, but in JSON they must be double.

{ Let's code that

```
import json

sample_response = """{
    "station": "Meadows",
    "reading": {
        "time": "2026-11-19T09:45:00+00:00",
        "temperature_c": 7.2
    },
    "online": true
}"""

observation = json.loads(sample_response)

print(observation)
print(type(observation))
print(type(observation["reading"]["temperature_c"]))
print(type(observation["reading"]["time"]))
```

```
{'station': 'Meadows',
 'reading': {'time': '2026-11-19T09:45:00+00:00',
             'temperature_c': 7.2},
 'online': True}
<class 'dict'>
<class 'float'>
<class 'str'>
```

Note: `json.loads()` reads JSON from a string – loads means load from string.

> JSON and Python types

```
{'station': 'Meadows',
 'reading': {'time': '2026-11-19T09:45:00+00:00',
             'temperature_c': 7.2},
 'online': True}
<class 'dict'>
<class 'float'>
<class 'str'>
```

Note:

- 7.2 is interpreted by Python as of type `float`
- `true` (JSON type) is translate to `True` (Python type)
- The time remains a `string`.
  - This is because JSON has no separate date or time type, so if we want Python to treat this as a datetime object, we need to use `datetime.fromisoformat()` to translate it.

> Reading and writing with JSON files

Rather than using `json.loads()` to load a string, we can use `json.load()` to load a file.

We can create a file and write to it in the same way we saw earlier. We can then read from it by using `json.load()`.

{ Let's code that

```
from pathlib import Path

data_dir = Path("selfguided_output")
data_dir.mkdir(exist_ok=True)
json_path = data_dir / "weather_observation.json"

with open(json_path, "w", encoding="utf-8") as file:
    file.write(sample_response)

with open(json_path, "r", encoding="utf-8") as file:
    saved_observation = json.load(file)

print(saved_observation == observation)
```

> Summary

- We can use the `datetime` module to work with dates and times
- `timedeltas` represent durations
- Naïve datetimes only work with respect to a given time zones, but aware datetimes give time zone information and are valid anywhere
- Asynchronicity can be to do things concurrently – i.e., interweaving two activities rather than waiting for one to finish before you start the next
- JSON is a widely used text format for storing and exchanging data

> What now??

No more new content!

Use the slides, recordings, self-guided notebooks and extra material to prepare for upcoming assignment quizzes and class tests:

- Assignment quiz – Friday Week 10
- Class Test – Thursday Week 11
- Final catch assessment (quizzes and/or class tests) – Thursday Week 12 (plus alternative for those with exam clashes

> What now??

If you complete all parts by the end of semester:

- Well done – you have learnt the full course!

If you have passed A1 and CT1 but haven't finished all the parts:

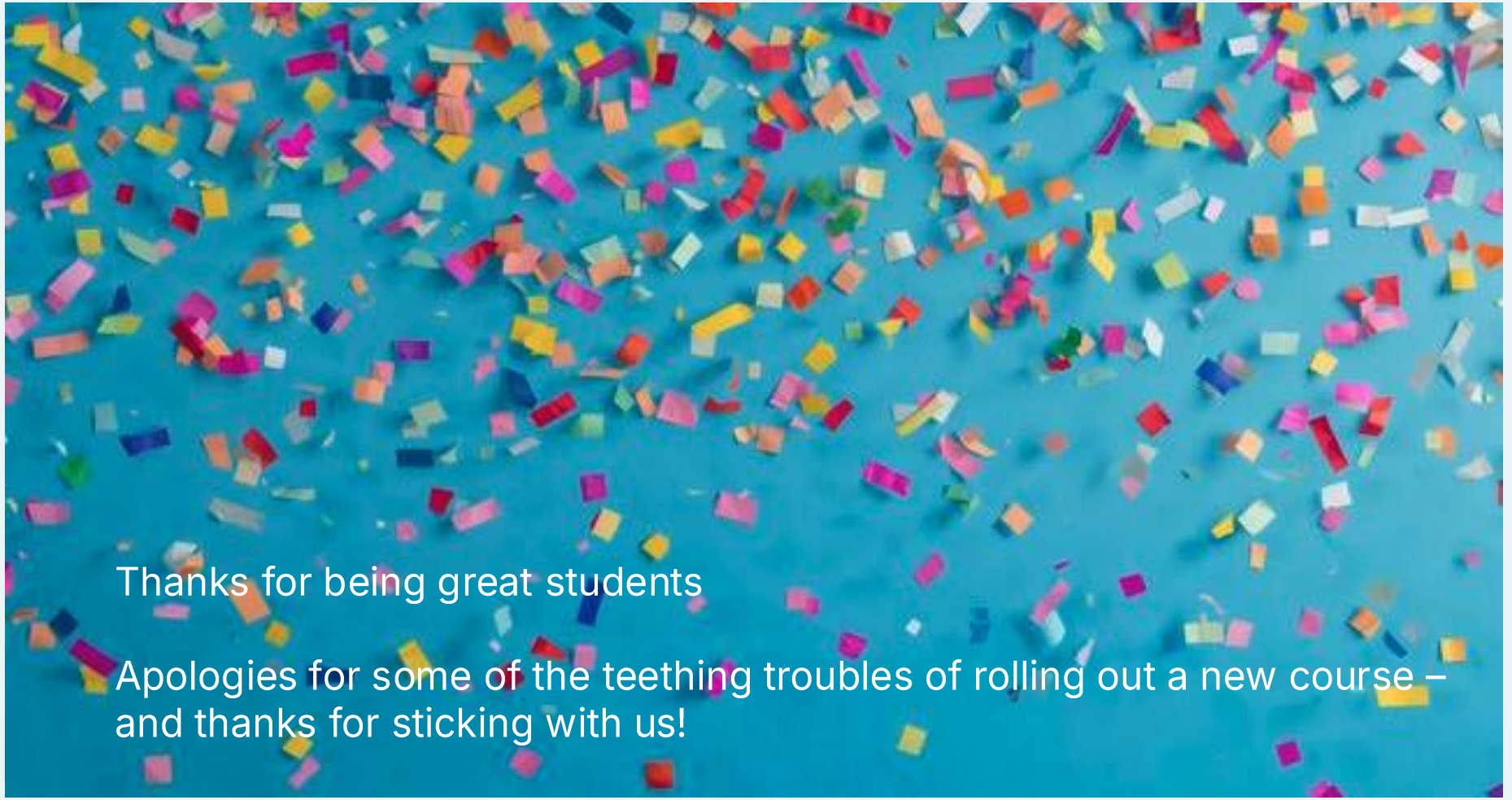
- Congratulations – you have passed the course
- But there are bits you've not learnt. This is normal and fine and lecturers in later years will know what is core in IP (what everyone has learnt) and what isn't (what some but not all have learned).
- You will continue to have access to IP material, and you can use this to help you brush up on bits you've not finished over the summer, in later courses when you need it, etc.

> What now??

If you have still not passed either or both A1 of CT1 after the four assessment points:

- You will be offered a retake in the summer. This will be similar to assessment during semester. We will send more info about this nearer the time
- If you are an Informatics student, you must pass this course, either during semester or in the summer retake diet. If you don't, you will need to retake the year.
- But there are lots of opportunities to practice your skills so that you do!

> What now??


Thanks for being great students
Apologies for some of the teething troubles of rolling out a new course – and thanks for sticking with us!