In [1]:
```
from cs103 import *
```

CPSC 103 - Systematic Program Design

Module 07a Day 2

Rik Blok, with thanks to Giulia Toti

Reminders

- Today: Deadline for final exam conflict notifications
- Today-Fri: Module 7 Tutorial Attendance
  - **Wed tutorial section T1T may attend any Thu-Fri section this week**
- Fri: Module 5 (Arbitrary Sized): Tutorial Resubmission
- next Wed-Fri: Module 7 Week 2 Tutorial Attndance

> ⚠ **Project Milestone**
>
> **DUE:** Friday Nov 24
>
> Start working on it now! Watch for your mentor TA's office hours to get help.
>
> You do not need to design `main` and `analyze` for the milestone.
>
> Review the grading rubric carefully so that your submission fulfills our requirements!
>
> **Syzygy may crash occasionally or become very slow as the deadline approaches, due to the volume of activity. Be sure to start early.**
>
> Note that office hours and Piazza will get busy so stay ahead of the curve!

See your Canvas calendar (https://canvas.ubc.ca/calendar) for details.

How to Design Analysis Programs (HtDAP)

The steps in the HtDAP recipe are:

1. Planning
   a. Plan input: Identify the information in the file your program will read.
   b. Plan output: Write a description of what your program will produce.
   c. Plan examples: Write or draw examples of what your program will produce.
2. Designing the program
   a. Design data definitions.
   b. Design read function: Design a function to read the information and store it as data in your program.
   c. Design analyze function: Design functions to analyze the data.

Example: Analysing VPD Crime Data

Step 1. Planning - Highlights

Step 1a. Plan input: Identify the info your program will read

- TYPE: The type of crime activities. One of
  - BNE Commercial
  - BNE Residential/Other
  - Theft of Vehicle
  - Theft of Bicycle
- HOUR,MINUTE: when the reported crime activity occurred
  - HOUR: A two-digit field that indicates the hour time (in 24 hours format)
  - MINUTE: A two-digit field that indicates the minute
  - **Note: Some crimes may not contain time information. In this case the hour and minute fields will both be zero.**

Step 1b. Plan output: Write a description of what your program will produce

- Given a type of crime, find the time of day (hour) with the highest frequency

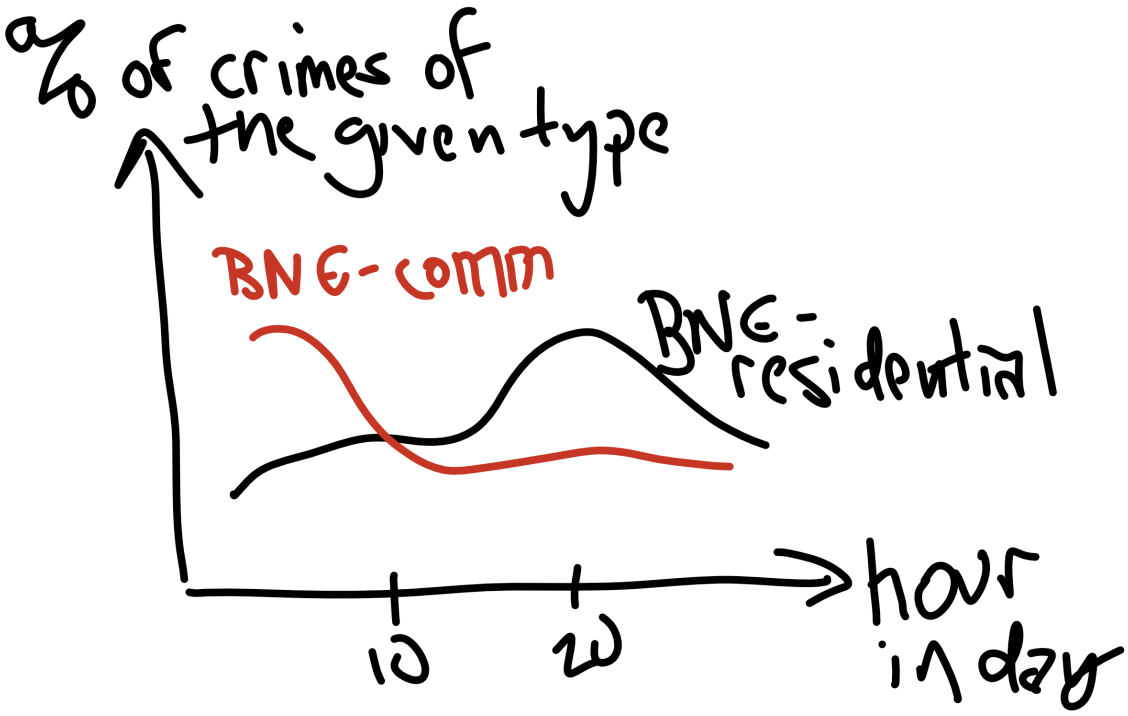
OR

- Graph of crimes committed in each hour, maybe show different types overlaid in different colours

Step 1c. Plan examples: Write or draw examples of what your program will produce

```
expect(main('crime_data_file.csv', CrimeType.BEC), 8)
```

OR



Step 2a: Design data definitions

Design data definitions:

- You should design data that your program is going to use.
- You should focus only on the fields/columns that your program is going to use and choose the correct type for each one.

- If you choose a non-primitive (interval, enumeration, optional...) you have to design it.
- Create a compound data to represent one line of data on the CSV file (`Consumed`).
- Create a list from this compound data to represent all lines of data in the CSV file (`List[Consumed]`).
- You may need more data that can be added later (for example, `List[int]` or `List[str]`).

Step 2a: Design data definitions - Example

Document which information you will represent in your data definitions

Before you design data definitions in the code cell below, you must explicitly document here which information in the file you chose to represent and why that information is crucial to the chart or graph that you'll produce when you complete step 2c.

✅ For this example, we'll skip the "chart or graph" because our program is producing a numerical quantity, instead (a particular hour of the day). We'll learn about graphs in the next module.

Put your answer here. Please don't delete the two HTML tags on either end of this paragraph. It's to make your answer blue so the TAs can easily spot it.

▸ ℹ️ Sample solution (For later. Don't peek if you want to learn 🙂)

Design data definitions

✅ Exam reference sheet

We recommend you print out the Exam reference sheet and use it as you're solving problems. The same sheet will be provided with your exams.

In [ ]:
```
from typing import NamedTuple, List
import csv

###################
# Data Definitions

Consumed = ...



# List[Consumed]
# interp. a list of Consumed

LOC0 = []

@typecheck
def fn_for_loc(loc: List[Consumed]) -> ...:
    ... # choose which template body to use for List[Consumed]

```

▸ ℹ️ Sample solution (For later. Don't peek if you want to learn 🙂)

Step 2b: Design `read` function

Design a function to read the information and store it as data in your program:

- You should complete the `read` function from its template.
- Change the `Consumed` type name.
- Check what columns from the file you need.
- Check if the types need parsing (changing the representation on the computer). All values in the CSV file are represented as strings when read.
- You can also add any other helper function you need (e.g. to remove rows with missing/invalid data).
- You should create at least two small CSV files for testing, so you can be sure your function is working before using it on full datasets.

> ⚠ You may skip irrelevant columns and rows with missing/invalid data when reading. But **don't remove any other rows**. (E.g., don't filter for a specific CrimeType here.) That's part of the chosen analysis and should be handled later. At this point we want to read **all** valid, relevant data.

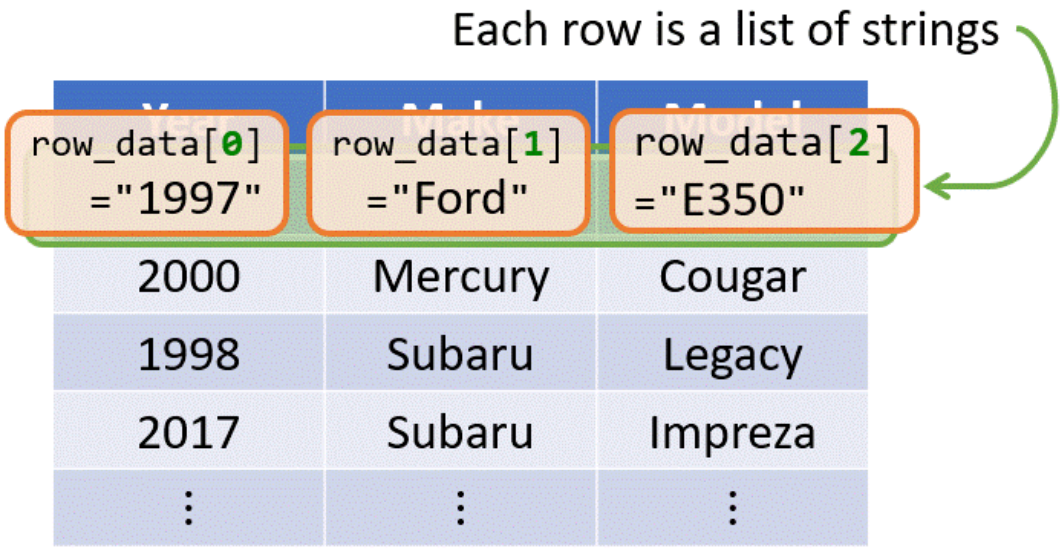
Reading rows from the CSV

From the HtDAP template:

```
@typecheck
def read(filename: str) -> List[Consumed]:
    ...
    with open(filename) as csvfile:

        reader = csv.reader(csvfile)
        next(reader) # skip header line

        for row_data in reader:
            # you may not need to store all the strings in the
            # current row, and you may need to convert some
            # of the strings to other types
            c = Consumed(row_data[0], ... ,row_data[n])
        ...
```



Reading a CSV file

⚠ Note that `row_data[n]` represents the **nth string in the current row**, not the nth row!

---

iClicker question: Which strings in `row_data`?

Open the crimedata file and have a look at the data. Which strings in `row_data` will represent the type of crime and the hour, respectively?

A. `row_data[0]` and `row_data[1]`
B. `row_data[1]` and `row_data[2]`
C. `row_data[0]` and `row_data[4]`
D. `row_data[1]` and `row_data[5]`
E. `row_data[5]` and `row_data[1]`

▶ ℹ Hint (for your review after class)

Parsing a row

The CSV reader will represent all items in a row as strings. We need that data to be *parsed* (converted) into the data types we're using. For example, for our crime data we want the hour to be stored as an integer.

The cs103 library contains two functions to parse data into integers and floats. From their help (e.g., `help(parse_int)`):

- `parse_int(s: str) -> Optional[int]` return `s` as an integer, if possible; returns `None` if `s` is not an integer
- `parse_float(s: str) -> Optional[float]` return `s` as a float, if possible; returns `None` if `s` is not a float

Parsing non-primitive data

For other, non-primitive data, you need to create your own `parse_my_data_type` helper function that converts a string into an instance of your data type.

Example

If we wanted to store the first four columns into a compound data type `Consumed` containing a string, integer, float, and `CrimeType` then our `for` loop would look like:

```
for row_data in reader:
    c = Consumed(row_data[0],
                 parse_int(row_data[1]),
                 parse_float(row_data[2]),
                 parse_crime_type(row_data[3]))
    loc.append(c)
```

iClicker question: Creating an instance from a row

We're trying to read the CSV file on the right into this data definition:

Name	GPA	StID
Jane Doe	3.9	12345678
⋮	⋮	⋮

```
Student = NamedTuple('Student',
                     [('name', str),
                      ('id', int),
                      ('gpa', float)]
```

We've written the following code to loop through the CSV file:

```
for row_data in reader:
    s = ...
    los.append(s)
```

Which of the following correctly fills in the missing expression (replacing the `...` ellipsis) to create an instance of `Student` ?

A. `Student(row_data[0], parse_int(row_data[2]), parse_float(row_data[1]))`
B. `Student(row_data[0], parse_float(row_data[2]), parse_int(row_data[1]))`
C. `Student(row_data[0], parse_int(row_data[1]), parse_float(row_data[2]))`
D. `Student(row_data[0], parse_float(row_data[1]), parse_int(row_data[2]))`

► ℹ Hint (for your review after class)

Step 2b: Design `read` function - Example

Space reserved for helper function

Continue on to the next cell to design `read` . We'll come back here later when we need to design a helper function.

In [ ]:
```
1
2
3
```

Design a function to read the information and store it as data in your program

The `parse_crime_type` helper has already been designed below. Let's jump down to that cell to run it and then return here. (For smoother flow, we put the helper below, even though it will break the kernel's "Restart & Run All" functionality.)

We also have example files and expected results prepared below. When we need it we'll jump down again to copy that example data.

In [ ]:

```
@typecheck
def read(filename: str) -> List[Consumed]:
    """
    reads information from the specified file and returns ...
    """
    #return []  #stub
    # Template from HtDAP
    # loc contains the result so far
    loc = [] # type: List[Consumed]

    with open(filename) as csvfile:

        reader = csv.reader(csvfile)
        next(reader) # skip header line

        for row_data in reader:
            # you may not need to store all the strings in the
            # current row, and you may need to convert some
            # of the strings to other types
            c = Consumed(row_data[0], ... , row_data[n])
            loc.append(c)

    return loc


start_testing()

# Examples and tests for read
expect(..., ...)

summary()

```

▶ ℹ️ Sample solution (For later. Don't peek if you want to learn 🙂)

Tests

Don't use the original CSV file to test your program. Otherwise you have to know the expected output before you've finished designing the program. (Isn't this what the program is for? 😉)

Instead, create special test files that contain a subset of the data. Choose the subsets so that:

1. They cover a wide range of data,
2. They cover special cases and edge cases,
3. They deliberately include some missing/invalid data that you think the program should be able to handle,
4. It's easy to determine what the expected output should be.

Small files are fine! You might also want one or two tests with larger subsets.

Give the test files names that help you remember what they're testing. Feel free to construct artificial data or sample from the original CSV file.

ℹ️ Don't forget to submit all your test files with your program!

Appendix: Parsing `CrimeType` with

parse_crime_type

Here's the helper function `parse_crime_type` to parse a string into a `CrimeType`. It was designed using our good old HtDF recipe. (The function wasn't designed in class because we're all familiar with the recipe by now 🙂 We just need to run this cell.)

In [ ]:

```
@typecheck
def parse_crime_type(s:str) -> CrimeType:
    """
    Returns the string s as a CrimeType.

    Assumes s is one of the following:
        "Break and Enter Commercial"
        "Break and Enter Residential/Other"
        "Theft of Bicycle"
        "Theft of Vehicle"
    """
    # return CrimeType.BEC # stub
    # template from atomic non-distinct
    # return ...(s)
    if s == "Break and Enter Commercial":
        return CrimeType.BEC
    elif s == "Break and Enter Residential/Other":
        return CrimeType.BER
    elif s == "Theft of Bicycle":
        return CrimeType.TB
    elif s == "Theft of Vehicle":
        return CrimeType.TV


start_testing()

expect(parse_crime_type("Break and Enter Commercial"), CrimeType.BEC)
expect(parse_crime_type("Break and Enter Residential/Other"), CrimeType.BER)
expect(parse_crime_type("Theft of Bicycle"), CrimeType.TB)
expect(parse_crime_type("Theft of Vehicle"), CrimeType.TV)

summary()

```

Now we can go back up to designing the `read` function and use this helper as needed.

Appendix: Parsed test data

Here are the test files parsed into `List[CrimeData]`. I've included this info here so we can quickly add it as needed to our examples.

In [ ]:

```
# from 'testfile_all_bec.csv'
TEST_ALL_BEC = [CrimeData(CrimeType.BEC, 6),
                CrimeData(CrimeType.BEC, 18)] # missing data removed

# from 'testfile_all_ber.csv'
TEST_ALL_BER = [CrimeData(CrimeType.BER, 21),
                CrimeData(CrimeType.BER, 17),
                CrimeData(CrimeType.BER, 0)] # reliable because nonzero minute field

# from 'testfile_all_tb.csv'
TEST_ALL_TB = [CrimeData(CrimeType.TB, 1),
               CrimeData(CrimeType.TB, 23),
               CrimeData(CrimeType.TB, 17)]

# from 'testfile_all_tv.csv'
TEST_ALL_TV = [CrimeData(CrimeType.TV, 23),
               CrimeData(CrimeType.TV, 14),
               CrimeData(CrimeType.TV, 21)]

# from 'testfile_all_missing.csv'
TEST_ALL_MISSING = [CrimeData(CrimeType.BEC, 0),
                    CrimeData(CrimeType.BER, 0),
                    CrimeData(CrimeType.TB, 0),
                    CrimeData(CrimeType.TV, 0)]

# from 'testfile_empty.csv'
TEST_EMPTY = []

```

We'll run the cell here to create these instances and copy the data to help prepare our examples above.