

# 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](#) next week**
- Fri: Module 5 (Arbitrary Sized): Tutorial Resubmission
- next Wed-Fri: Module 7 Week 2 Tutorial Attendance

#### [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

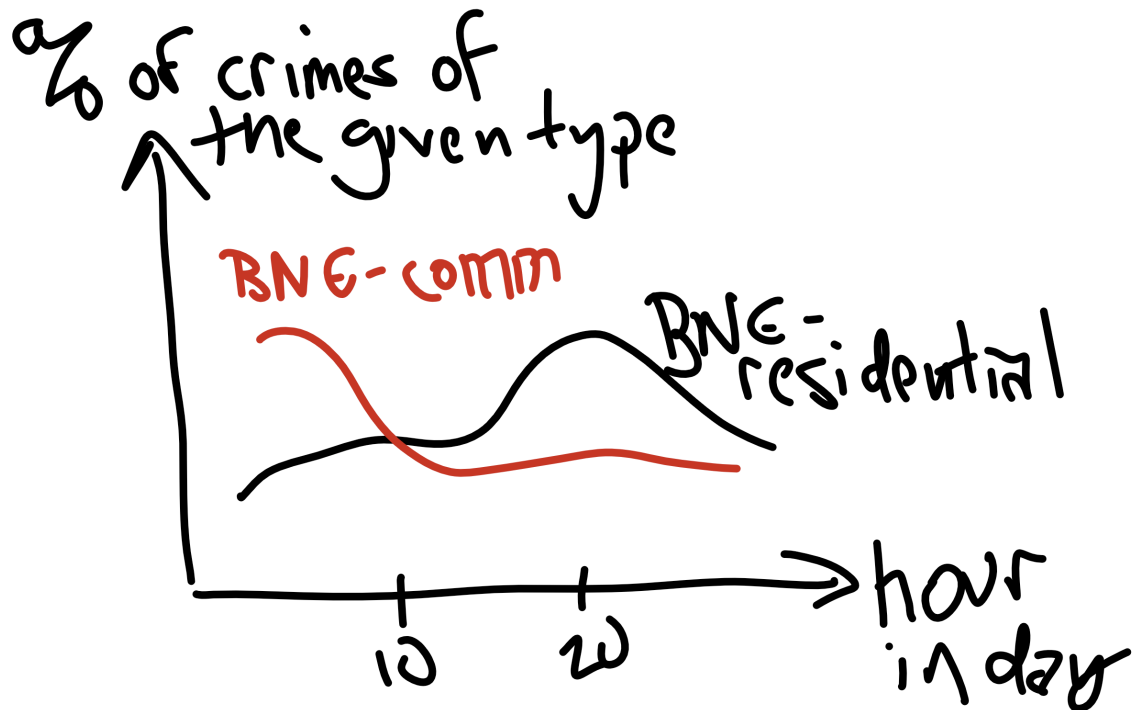
OR

- 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.

*Note: we'll skip the "chart or graph" part!*

✓ 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).

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.

▼ ⓘ Sample solution (For later. Don't peek if you want to learn 😊)

#### Which information you chose to represent:

We are only interested in the Type and Hour columns. Type can be represented as an Enum (4 cases) and Hour as an integer interval [0,23].

We also need the Minute column to check if the time data is reliable but we don't need to store it.

### Design data definitions

```
In [ ]: 1 from cs103 import *
        2 from typing import NamedTuple, List
        3 import csv
        4
        5 #####
        6 # Data Definitions
        7
        8 Consumed = ...
        9
        10
        11
        12 # List[Consumed]
        13 # interp. a list of Consumed
        14
        15 LOC0 = []
        16
        17 @typecheck
        18 def fn_for_loc(loc: List[Consumed]) -> ...:
        19     ... # choose which template body to use for List[Consumed]
        20
        21
```

▶ ⓘ 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 needs 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])
            ...
```

Each row is a list of strings

Year	Make	Model
row_data[0] ="1997"	row_data[1] ="Ford"	row_data[2] ="E350"
2000	Mercury	Cougar
1998	Subaru	Legacy
2017	Subaru	Impreza
⋮	⋮	⋮

# Reading a CSV file


⚠ Note that `row_data[n]` represents the **n<sup>th</sup> string in the current row**, not the n<sup>th</sup> row!

## iClicker question: Which strings in row?



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]`

▶  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 `Consumed` was a compound 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	StdID
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]))

▶  Hint (for your review after class)

## Step 2b: Building - Example

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

```
In [ ]: 1 @typecheck
        2 def read(filename: str) -> List[Consumed]:
        3     """
        4     reads information from the specified file and returns ...
        5     """
        6     #return [] #stub
        7     # Template from HtDAP
        8     # loc contains the result so far
        9     loc = [] # type: List[Consumed]
       10
       11     with open(filename) as csvfile:
       12
       13         reader = csv.reader(csvfile)
       14         next(reader) # skip header line
       15
       16         for row_data in reader:
       17             # you may not need to store all the strings in the
       18             # current row, and you may need to convert some
       19             # of the strings to other types
       20             c = Consumed(row_data[0], ... , row_data[n])
       21             loc.append(c)
       22
       23         return loc
       24
       25
       26 start_testing()
       27
       28 # Examples and tests for read
       29 expect(..., ...)
       30
       31 summary()
       32
       33
```

►  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.

 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.)

Notice that this function takes a string as input, not a `CrimeType`. (That's the *output*!)

```

In [ ]: 1 @typecheck
2 def parse_crime_type(s:str) -> CrimeType:
3     """
4     Returns the string s as a CrimeType.
5
6     Assumes s is one of the following:
7     "Break and Enter Commercial"
8     "Break and Enter Residential/Other"
9     "Theft of Bicycle"
10    "Theft of vehicle"
11    """
12    # return CrimeType.BEC # stub
13    # template from atomic non-distinct
14    # return ...(s)
15    if s == "Break and Enter Commercial":
16        return CrimeType.BEC
17    elif s == "Break and Enter Residential/Other":
18        return CrimeType.BER
19    elif s == "Theft of Bicycle":
20        return CrimeType.TB
21    elif s == "Theft of vehicle":
22        return CrimeType.TV
23
24
25 start_testing()
26
27 expect(parse_crime_type("Break and Enter Commercial"), CrimeType.BEC)
28 expect(parse_crime_type("Break and Enter Residential/Other"), CrimeType.BER)
29 expect(parse_crime_type("Theft of Bicycle"), CrimeType.TB)
30 expect(parse_crime_type("Theft of vehicle"), CrimeType.TV)
31
32 summary()
33
34

```

## 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 [ ]: 1 TEST_ALL_MISSING = [CrimeData(CrimeType.BEC, 0),
2                             CrimeData(CrimeType.BER, 0),
3                             CrimeData(CrimeType.TB, 0),
4                             CrimeData(CrimeType.TV, 0)]
5
6 TEST_ALL_BEC = [CrimeData(CrimeType.BEC, 6),
7                  CrimeData(CrimeType.BEC, 18)] # missing data removed
8
9 TEST_ALL_BER = [CrimeData(CrimeType.BER, 21),
10                 CrimeData(CrimeType.BER, 17),
11                 CrimeData(CrimeType.BER, 0)]
12
13 TEST_ALL_TB = [CrimeData(CrimeType.TB, 1),
14                 CrimeData(CrimeType.TB, 23),
15                 CrimeData(CrimeType.TB, 17)]
16
17 TEST_ALL_TV = [CrimeData(CrimeType.TV, 23),
18                 CrimeData(CrimeType.TV, 14),
19                 CrimeData(CrimeType.TV, 21)]
20
21

```



