In [2]:
```
from cs103 import *
```

CPSC 103 - Systematic Program Design

Module 05a Day 2

Ian Mitchell, with thanks to Rik Blok and Giulia Toti

Reminders

- Wed: Module 5 Part 2: Pre-Lecture Assignment.
- Wed: Module 3 (HtDD): Tutorial Resubmission (optional).
- No tutorial due (but recommend you try to finish tutorial 5 before the midterm).
- No code review due (but you should still practice).
- No worksheet due (but you should still practice).
- **this Wed-Fri** (was Fri-Thu): Module 5 Week 2 Tutorial Attendance.
- **Next week Fri**: Midterm exam 6:30pm - 8:00pm.

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

iClicker check-in

Have you printed out a paper copy of the exam reference sheet and practiced some problems using it?

A. What is an "exam reference sheet"?
B. No, I've been just looking up HtDF, HtDD and Data Driven Templates online when I need them.
C. Yes, but I'm still looking up stuff online too.
D. Yes, and I don't need to look up anything else about HtDF, HtDD or Data Driven Templates as long as I have the sheet.

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

From last time

- Data definition for `List[str]` .
- Function definition for `count_UBCs()` .
- Function definition for `contains_UBC()` .

In [ ]:
```
from typing import List

# List[str]
# interp. a list of strings
LOS0 = []
LOS1 = ["hello", "goodbye", "Beatles"]

@typecheck
# template based on arbitrary-sized
def fn_for_los(los: List[str]) -> ...:
    # description of the accumulator
    acc = ...   # type: ...

    for s in los:
        acc = ...(s, acc)

    return ...(acc)
```

In [5]:
```
@typecheck
def count_UBCs(unis: List[str]) -> int:
    """
    return the number of times that "UBC" appears in unis.
    """
    #return -1 # stub

    # template from list of str.
    # count_ubc is the number of times we have seen "UBC" in unis so far.
    count_ubc = 0   # type: int

    for uni in unis:
        if uni == "UBC":
            count_ubc = count_ubc + 1

    return count_ubc

start_testing()

# Here are some examples. Do we need any more?
expect(count_UBCs(["UW"]), 0)
expect(count_UBCs(["UBC"]), 1)
expect(count_UBCs(["McGill", "SFU", "UBC", "MIT", "Harvard"]), 1)
expect(count_UBCs(["McGill", "SFU", "MIT", "Harvard"]), 0)
expect(count_UBCs(["UBC", "McGill", "SFU", "UBC", "MIT", "Harvard", "UBC"]), 3)
# We needed to add the empty list as an example.  Generally speaking, we encourage
# to make the empty list your first test case so you don't forget.  But we'll put i
# to make clear that it was an extra test case missing from the partially completed
expect(count_UBCs([]), 0)

summary()
```

```
6 of 6 tests passed
```

In [7]:
```
# Constructed from answer to clicker question, but had to add some tests.
# This version didn't need an accumulator.

@typecheck
def contains_UBC(unis: List[str]) -> bool:
    """
    Return True if unis includes "UBC", otherwise return False.
    """
    # return True # stub
    # Template based on List[str]
    for u in unis:
        if u == "UBC":
            return True
    return False

expect(contains_UBC(["UW"]), False)
expect(contains_UBC(["UBC"]), True)
expect(contains_UBC(["McGill", "SFU", "UBC", "MIT", "Harvard"]), True)
expect(contains_UBC(["McGill", "SFU", "MIT", "Harvard"]), False)
expect(contains_UBC(["UBC", "McGill", "SFU", "UBC", "MIT", "Harvard", "UBC"]), True
# We needed to add the empty list as an example.  Generally speaking, we encourage
# to make the empty list your first test case so you don't forget.  But we'll put i
# to make clear that it was an extra test case missing from the partially completed
expect(contains_UBC([]), False)
```

Exercise

If we've already designed `count_UBCs`, can we use that function to implement `contains_UBC`? As an exercise, redesign the function body of `contains_UBC` to call `count_UBCs`.

In [ ]:
```
# TODO: Redesign the body of contains_UBC to call count_UBCs
```

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

In programming, redesigning the implementation of a function or other collection of code **without** changing its external behaviour is called *refactoring*. In the case of functions, the external behaviour is defined by the signature and purpose, so if we change the body without changing the signature and purpose then we are refactoring. In the exercise above, we refactored `contains_UBC()`.

Refactoring is often necessary in any program which sees long-term use; for example, refactoring can be done to reduce the execution time or memory that a program requires (*optimization* of the code), or to set things up so that it is easier to add new features to the program in the future.

But refactoring is also one of the most common ways in which bugs sneak into a program; consequently, it is a time when your previously written tests demonstrate their continuing value: You can run them on the refactored code (called *regression* testing) to see if you accidentally broke anything. Regression testing is used in other circumstances, but is critical to quickly building confidence in refactored code.

Designing a data definition for a list of integers

Problem: design a data definition for a list of integers.

Once we've learned how to design other data definitions and realize that a data definition for a particular problem needs to be a list, these tend to be fairly straightforward to complete. Let's practice one quickly this week.

Next week, we'll see how things change (for lists, optionals, and compounds!) when a data definition we create refers to another data definition we created!

✅ Exam reference sheet We recommend you print out the [Exam reference sheet](https://canvas.ubc.ca/courses/123409/files/27906280?module_item_id=5918654) and use it as you're solving problems. The same sheet will be provided with your exams.

In [ ]:
```
# TODO: Design a data definition for a list of integers
```

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

Accumulator

Recall, an *accumulator* is used in the template for list data definitions:

- Stores useful data about our progress through the list
- No special meaning to Python, just another variable
- You can rename it from `acc` to something more meaningful
- You can create multiple accumulator variables, if necessary

---

Exercise 1: Find the largest integer

Problem: Find the largest integer in a list.

iClicker check-in

Wait a minute!

A. We already saw this problem in the Module 5 Screencasts, so why should we do it again?
B. We already saw this problem in the Module 5 Screencasts, but I don't think that answer was correct.
C. We already saw this problem in the Module 5 Screencasts, but I don't think that answer was complete.
D. Nevermind. This looks like a nice new problem for me to practice on.

In [ ]:
```
# TODO: Follow the HtDF recipe to write a function that finds the largest integer i
```

▸ ⚠️ Things to watch out for! (Read after completing your solution.)

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

Exercise 2: Find the even numbers

Problem: Find all the even numbers in a list.

First, let's brainstorm an outline to the solution.

Two subproblems

In solving this problem, you can expect to encounter two subproblems:

1. How do you check if a number is even?
2. How do you attach a new item to the end of a list?

Let's solve each of these subproblems now.

ℹ️ Tip 1: `%` (modulo) operator From the [Python Language Reference](https://docs.python.org/3/reference/expressions.html#binary-arithmetic-operations): > The `%` (modulo) operator yields the remainder from the division of the first argument by the second. **Examples** | Expression | Value | Reason | In other words | |----------:|:------|:------:|:---------------:| | `4 % 2` | `0` | 4 divided by 2 is 2 with remainder 0 | `4 == 2*2 + 0` | | `5 % 2` | `1` | 5 divided by 2 is 2 with remainder 1 | `5 == 2*2 + 1` | | `6 % 2` | `0` | 6 divided by 2 is 3 with remainder 0 | `6 == 3*2 + 0` | | `6.5 % 2` | `0.5` | 6.5 divided by 2 is 3 with remainder 0.5 | `6.5 == 3*2 + 0.5` |

iClicker question: How to check if a number is even?

You want to determine if a number `i` is even. Which expression will return `True` if and only if `i` is even?

A. `2 % i != 0`
B. `2 % i == 0`
C. `i % 2 != 0`
D. `i % 2 == 0`

▶ ℹ️ Hint (for your review after class)

ℹ️ Tip 2: Appending to a list The [Python Language Reference](https://docs.python.org/3/tutorial/datastructures.html#more-on-lists) tells us how to append (add to the end) items to an existing list: 1. We can join two lists with the `+` operator, or 2. We can append a single item with the `.append` method. **Examples** ```python los1 = ['one', 'two'] los2 = ['three', 'four'] item = 'three' # join two lists with the `+` operator los1 + los2 # evaluates to ['one', 'two', 'three', 'four'] # append a single item to `los1` los1.append(item) # updates `los1` to be ['one', 'two', 'three'] ``` Try them out for yourself in the cell below!

In [ ]:
```
los1 = ['one', 'two']
los2 = ['three', 'four']
item = 'three'

# Try them out here!
```

iClicker question: How to append an item to a list?

```
los1 = ['one', 'two']
item = 'three'
```

You want to update the list `los1` above to contain `['one', two', 'three']`. Which expression will perform the task? Select ALL that apply. [Set iClicker question type to "Multiple Answer".]

A. `los1 + item`
B. `los1 + [item]`
C. `los1 = los1 + item`
D. `los1 = los1 + [item]`
E. `los1 = los1.append(item)`

▸ ℹ Hint (for your review after class)

Back to Exercise 2: Find the even numbers

Problem: Find all the even numbers in a list.

Let's complete Steps 1–3 (**S**tub, **E**xamples, and **T**emplate) of the HtDF recipe together. Then you'll finish the remaining steps on your own.

In [ ]:
```
# TODO: Your program goes here
```

▸ ℹ Sample solution of HtDF Steps 1–3 (For later. Don't peek if you want to learn 🙂)

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

iClicker question: Target a bug

The following program was designed to calculate the product of all numbers in a list of integers... but it contains a few "bugs" (errors). Tap on one of the bugs you find. (Examples are hidden and can be considered correct.) [Set iClicker question type to "Target".]

In [ ]:
```python
@typecheck
def product(loi: List[int]) -> int:
    """
    Return the product of all integers in loi
    """
    # return 0 # stub

    # template from List[str]
    prod = 0 # type: int

    for i in loi:
        prod = prod * loi

    return prod


# examples are hidden and can be considered correct
```

▶ ℹ Hint (for your review after class)

▶ ℹ Corrected program (For later. Don't peek if you want to learn 🙂)

Exercise 3: Fun with a Large-ish Information Set

The data below represents the length of episodes from a TV show, in minutes. We have examples for only one episode of a show (Friends), a full season of a show (Game of Thrones), and a whole show (The Good Place).

ℹ Things to notice - We've chosen to name our data type `EpisodeDurations` because it's more meaningful than just `List[float] # in range [0, ...]` - Likewise, we've given our loop variable a descriptive name, `duration` - In Python a loop variable (e.g., `duration`) [persists](https://docs.python.org/3/reference/compound_stmts.html#the-for-statement) after the loop finishes. Not the case for all languages

In [ ]:
```
from typing import List

EpisodeDurations = List[float] # in range [0, ...]
# interp. the duration of episodes in minutes for some number of
# episodes of a TV Show

ED0 = []
ED_FRIENDS_S01E01 = [22.8]
ED_GAME_OF_THRONES_S01 = [61.62, 55.28, 57.23, 55.62, 54.27, 52.6,
                          57.79, 58.13, 56.27, 52.62]
ED_GOOD_PLACE = [
    26.27, 21.50, 24.90, 22.55, 26.30, 26.35, 24.23, 25.23, 24.88,
    23.78, 26.62, 21.53, 26.88, 42.68, 21.60, 23.92, 25.37, 24.65,
    23.28, 23.72, 21.60, 24.78, 22.77, 23.47, 24.33, 21.60, 21.55,
    21.60, 21.60, 21.60, 21.53, 21.55, 21.53, 21.53, 22.53, 21.53,
    21.53, 21.53, 22.42, 21.40, 21.42, 21.43, 21.43, 21.42, 21.42,
    21.40, 21.42, 21.42, 21.45, 21.43, 52.48
]

# template based on arbitrary-sized
@typecheck
def fn_for_episode_durations(ed: EpisodeDurations) -> ...:
    # description of the accumulator
    acc = ...   # type: ...

    for duration in ed:
        acc = ...(duration, acc)

    return ...(acc)
```

Now, let's design a function that finds the average duration (in minutes) of all episodes in an `EpisodeDurations` list.

We're going to end up using multiple accumulators in this design. Template functions in data definitions give you code that *may* be useful to you, but you might decide not to use some pieces, to add or duplicate pieces, or to insert code that isn't suggested by the template at all.

For the list template, the accumulator is a suggestion: you may not need one, you may need one, or in cases where you're tracking multiple different evolving pieces of information through the loop, you may need multiple!

In [ ]:
```
# We've completed signature, purpose, stub, and examples.
# Let's continue from there!

@typecheck
def avg_episode_duration(ed: EpisodeDurations) -> float:
    """
    Return the average duration (in minutes) of the episodes in ed.

    (The average duration of zero episodes is returned as 0.)
    """
    return 0.0  #stub

start_testing()

expect(avg_episode_duration([]), 0.0)
expect(avg_episode_duration([1, 1, 1, 1, 1]), 1)
expect(avg_episode_duration([100.12, 12.1]), (100.12+12.1)/2)
expect(avg_episode_duration(ED_GAME_OF_THRONES_S01),
       (61.62+55.28+57.23+55.62+54.27+52.6+57.79+58.13+56.27+52.62)/10
      )
expect(avg_episode_duration(ED_FRIENDS_S01E01), 22.8)

summary()
```

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

In [ ]:
```
# Now, what we all came here for; the average episode length of The Good Place:
"The Good Place is the smartest, dumbest " + str(avg_episode_duration(ED_GOOD_PLACE
```

iClicker question: Do you still have questions?

Here are some common questions that came up in this week's pre-class assignment. Which questions are you still unsure about? Select as many as you like. [Set iClicker question type to "Multiple Answer".]

A. When do we use/not use the accumulator? And how do we indicate if it isn't being used?
B. What is the advantage of using lists vs other data structures? E.g., Enum or Compound.
C. What does the `n in for n in lon` mean? And how do we name these appropriately?
D. Nah, I'm pretty sure I can answer these questions. 🙂

▶ Hints (for your review after class)

Some Common Questions from Prior Years of CPSC 103

These questions from the pre-lecture assignment did not come up as commonly this year as those above, but they were common in prior years so we repeat them here in case you are still puzzled.

Q: When do we need more than one accumulator? A: We just saw an example, when we needed to accumulate two pieces of data in order to compute an average. Generally, think about how much data you need to store as you iterate through the items in the list.

Q: Can a function take in a list as a parameter without creating a data definition? A: Python does not **need** an HtDD data definition to work with a list (or an Enum, an Optional, a Compound, ...), but that wouldn't follow good programming practices, would it?

Q: What are loops? How do we use them? And when do we need them? A: Practice tracing through some example loops to study how they perform their tasks. Do each by hand first, then check that you got it right by tracing it again in PythonTutor. If you're still having trouble, take advantage of the course's resources to get more help: tutorials, Piazza, office hours, ...