```
In [ ]:  1  from cs103 import *
         2
         3
```

CPSC 103 - Systematic Program Design

Module 05b Day 1

Rik Blok, with thanks to Ian Mitchell and Giulia Toti

Reminders

- **Today-Tomorrow**: Module 5 Week 2 Tutorial Attendance
- Wed: Module 4 (Compound): Tutorial Resubmission (optional)
- next Wed-Fri: Office hours in tutorials (no attendance taken)
- next Fri: Midterm exam scheduled for Friday Oct 27 @ 6:30pm

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

Recap: Applying the HtDD recipe to lists

Data definition

- Can name (e.g., `ListOfDataType = List[DataType]`) but usually redundant
- Instead we often just add a comment that our new data type is a

  ```
  # List[DataType]
  ```

Interpretation

- This suffices:

  ```
  # interp. a list of DataType
  ```

- Can add more information if you know more about the data being stored
- But don't need to interpret the data type in the list... user can review its interp. statement

Examples

- Always consider the possibility that the list could be empty, `[]` .
- Naming convention: abbreviation of `List[DataType]` is `LODT` (for "list of data type")

Template

- function and parameter naming also from "list of data type":

  ```
  @typecheck
  # template based on arbitrary-sized
  def fn_for_lodt(lodt: List[DataType]) -> ...:
  ```

- accumulator variable (`acc`) helps us keeps track of something interesting we're looking at in the list:

```python
    # description of the accumulator
    acc = ...    # type: ...
```

- `for` loop iterates through all items in the list, performing the function's task
- update accumulator as we progress:

```python
    for dt in lodt:
        acc = ...(dt, acc)
```

- return something that depends on the final data stored in the accumulator

```python
    return ...(acc)
```

iClicker question: Appending to a list

The Python Language Reference 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.

What is the final value of `los` after the program to the right has executed?

A. `['three', 'four']`
B. `['one', 'two', 'three']`
C. `['one', 'two', 'three', 'three']`
D. `['one', 'two', 'three', 'four']`
E. The program won't complete because it contains an error

```python
los = ['one', 'two']
item3 = 'three'
item4 = 'four'

los = los + [item3]
los.append(item3)
```

▶ ℹ️ Hint (for your review after class)

Arbitrary-Sized Data (Lists) of Compounds

Compound data:

- Use when the information to be represented has two or more values that naturally belong together. E.g.,

    V1 = Velocity(9, 22)

Arbitrary sized data (List):

- A data type that has a group of elements of the same type
- It is very natural to want to create a list of compound data! E.g.,

    List[Velocity]

- When creating examples, include some that use the compound's examples. E.g.,

    LOV2 = [V1, V2]

- You also must use the Compound data template in the list template (reference rule) as we'll demonstrate below

Concepts covered

When data definitions contain other data definitions we need multiple functions to handle them. Adds a new "tier" of complexity to the course.

The reference rule

> ℹ️ **Reference rule:** When a data definition **refers** to other non-primitive data, delegate the operation to a helper function.

- Must be applied **every** time a data definition manipulates other data that is not primitive
- That includes List, Compound, Enumerated, and Optional types
- Anytime a variable is from a non-primitive type, we should invoke its template function. E.g., if `dt` is non-primitive then

```
# replace
return ...(dt)
# with
return ...(fn_for_data_type(dt))
```

- Add a comment to indicate the reference rule was applied:

```
# template based on ... and reference rule
```

Example 1: Reference rule

Problem: Given the compound data `Velocity` , write a function to compute the average speed of all velocities with a *northerly* heading in a list. We will consider directions in the range 45-135 degrees as northerly.

We'll need to tackle this problem in three stages:

1. Review the given `Velocity` data definition
2. Design a data definition for `List[Velocity]`
3. Design our function and any needed "helper" functions
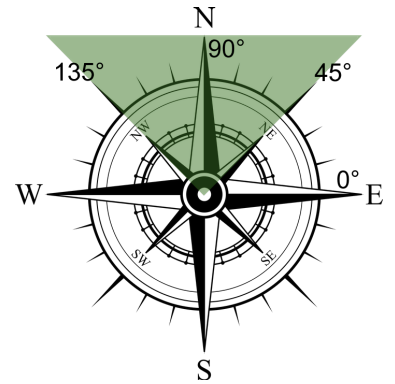
Stage 1: Review the given `Velocity` data definition

Recall our `Velocity` compound from Module 04 Day 1:

▶ Jump to...

```
In [ ]:  1  from typing import NamedTuple
         2  Velocity = NamedTuple('Velocity', [('speed', float),
         3                                     ('dir', int)]) # in range[0,359]
         4
         5  # interp. a velocity with its speed in m/s and direction
         6  # as an angle in degrees with east=0 increasing counterclockwise
         7
         8  V1 = Velocity(9, 22)
         9  V2 = Velocity(3.4, 180)
        10
        11  # template based on Compound (2 fields)
        12  @typecheck
        13  def fn_for_velocity(v: Velocity) -> ...:
        14      return ...(v.speed, v.dir)
        15
        16
```

Stage 2: Design a data definition for `List[Velocity]`

The **D**efinition, **I**nterpretation, **E**xamples, and our **usual T**emplate have already been completed. We still need to apply the reference rule.

▶ Jump to...

```
 1  from typing import List
 2  # List[Velocity]
 3  # interp. a list of velocities
 4
 5  LOV0 = []
 6  LOV1 = [Velocity(3.1, 41)]
 7  LOV2 = [V1, V2]
 8  LOV3 = [Velocity(5.9, 265), Velocity(3.5, 89), Velocity(7.9, 323)]
 9
10  @typecheck
11  # template based on arbitrary-sized
12  def fn_for_lov(lov: List[Velocity]) -> ...:
13      # description of the accumulator
14      acc = ... # type: ...
15      for v in lov:
16          acc = ...(v, acc)
17      return ...(acc)
18
19
```

iClicker question: Where to apply the reference rule?

We need to make two changes to the code above to apply the reference rule. Target either one of the places we'll need to change. [Set question type to "Target".]

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

Helper functions

- A *helper function* is a normal function, but instead of solving the main problem, it solves a small part of the problem, helping the main function to solve the problem
- The main function is the function that actually solves the problem and uses the helper function to achieve this
- A good design has several small helper functions that each do only a small task
- Every time the reference rule appears, it indicates that a helper function may be needed

Example: Top-down design

The `main` function here doesn't know how to work with non-primitive data `dt1` and `dt2` so it **refers** them to helper functions.

```
def main(dt1: DataType1, dt2: DataType2) -> ...:
    """
    The main function, which depends on two data types
    """
    return ...(fn_for_data_type_1(dt1), fn_for_data_type_2(dt2))
```

Notice we designed our `main` function **first** with calls to *helper function* stubs, then we design those helper functions **after**.

```python
def fn_for_data_type_1(dt1: DataType1) -> ...:
    """
    Helper function for DataType1
    """
    return ...  # stub

def fn_for_data_type_2(dt2: DataType2) -> ...:
    """
    Helper function for DataType2
    """
    return ...  # stub
```

Top-down vs. bottom-up approach
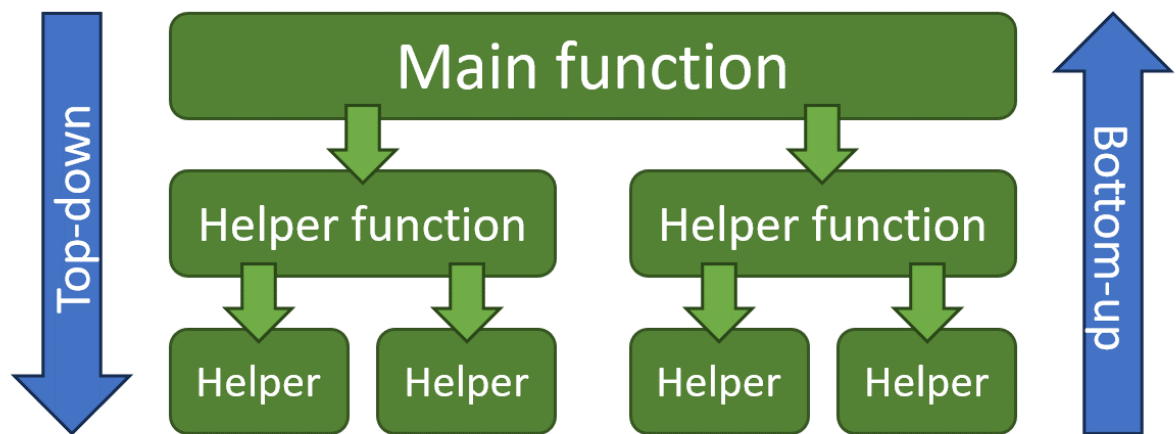
Top-down

- "Divide and conquer" strategy
- Tackle the *high-level* problem first
- Design your main function
    - Make calls to new helper functions as needed
    - Write stubs for the new helper functions as you go
- Then design your helper functions to solve the *low-level* subproblems

Bottom-up

- "Construction" strategy
- Tackle the *low-level* subproblems first
- Design helper functions to solve the subproblems
- Then design your main function – using the helper functions – to solve the *high-level* problem

iClicker question: Top-down vs. bottom-up

Ultimately, the choice of whether to take a top-down or bottom-up approach depends on the problem and your preferences/experience. You can use a combination of both.

Which of the following are advantages of the **top-down** approach? Select ALL that apply. [Set question type to "Multiple Answer".]

A. Facilitates teamwork by allowing tasks to be identified, divided up, and worked on simultaneously
B. Encourages focus on the big picture and long-term goals
C. Allows testing as code is developing
D. Helps to identify potential problems and solutions before implementation
E. Lets you jump straight into programming, without having to plan a solution to the full problem

> ▶ ℹ️ Hint (for your review after class)

> ⚠️ If your program spans multiple cells, **present** the code in a bottom-up style, with helper functions in cells **above** main program, **even if using top-down approach** (as we'll do in this lecture). This presentation order is needed if you want the program to run correctly on kernel restart (so that the lower level helper functions are defined before they are called by the higher level functions).

Decomposing the problem

Problem: Given the compound data `Velocity`, write a function to compute the average speed of all velocities with a northerly heading in a list.

Let's take a top-down approach

- "Divide and conquer" strategy
- Tackle the *high-level* problem first

Design a function `average_speed_of_northerly` that takes in `List[Velocity]` and returns the average speed of the Velocities with a northerly direction.

Subproblems

What subproblems do we need to solve to complete our solution?

> ▶ ✅ Answer (Don't peek if you want to learn 🙂)

iClicker question: Helper functions

Our task is now to design a function that takes in a list of Velocities and returns the average speed of the Velocities with a northerly direction.

Using the top-down design approach outlined above, it will iterate through the list of Velocities accumulating the sum of speeds and count of all northerly-directed Velocities in the list seen so far.

What helper functions will we need? Select ALL that apply. [Set iClicker question type to "Multiple Answer".]

A. A function that returns the maximum speed in a list of Velocities
B. A function that takes in a list of Velocities and returns the average speed of the Velocities with a northerly direction
C. A function that reports the direction of a Velocity
D. A function that reports whether a Velocity is northerly-directed
E. A function that reports the speed of a Velocity

> ℹ️ Hint (for your review after class)

Stage 3: Design our function and any needed "helper" functions

Problem: Given the compound data `Velocity`, write a function to compute the average speed of all velocities with a northerly heading in a list.

Approach to solution

Design a function `average_speed_of_northerly` that takes in `List[Velocity]` and returns the average speed of the Velocities with a northerly direction.

Example 1 Helper functions

We'll design our helper functions **after** the main function (top-down design) but we'll insert them above, here, so that the notebook runs correctly even after restarting the kernel. So we'll first skip down to the "Main function" section and start designing our function there.

> ▸ Jump to...

```
In [ ]:   1  # TODO: Design our helper functions here after designing main function, below
          2  # :
          3  # (how many helper functions do we need?)
          4
          5
```

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

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

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

Example 1 Main function

```
1  # TODO: Design our function
2
3
```

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

Reference rule applied to other data types

Lists can refer to other types defined in a data definition, but so can several other types of data. Specifically, Optionals and Compounds can refer to other data definitions. In those cases, you follow the same reference rule as with lists.

Every time you are manipulating non-primitive data, the reference rule applies. The template, if written correctly, will help you know when a function is needed.

In the case of a function for a `List[Velocity]` we applied the reference rule to refer to data of the non-primitive `Velocity` type. We didn't need to apply the reference rule to any functions for `Velocity` itself because it just refers to primitive data types. (We treat the interval `dir` as primitive).

$$\text{List} \quad \rightarrow \quad \text{Velocity} \quad \rightarrow \quad \text{float (primitive)}$$
$$\rightarrow \quad \text{int (primitive)}$$

Exercise: Repeat for a modified `Velocity`

Let's try working with a modified version of `Velocity` that contains a `Direction` as a cardinal compass direction (enumeration).

Now the compound refers to a non-primitive field:

$$\text{List} \quad \rightarrow \quad \text{Velocity} \quad \rightarrow \quad \text{float (primitive)}$$
$$\rightarrow \quad \text{Direction} \quad \rightarrow \quad \text{distinct value (primitive)}$$

Problem: Given the modified compound data `Velocity`, write a function to compute the average speed of all velocities with a *northerly* heading in a list. (All velocities with `Direction.N` are northerly.)

Let's see how we apply the reference rule for this modified data type.

The data definition for `Direction` is provided

```
1  from enum import Enum
2
3  Direction = Enum("Direction",["N", "E", "S", "W"])
4
5  #interpr. a direction (N - North, E - East, S - South, W - West)
6
7  # Examples are redundant for Enumeration
8
9  @typecheck
10 # template for Enumeration (4 cases)
11 def fn_for_direction(d: Direction) -> ...:
12     if d == Direction.N:
13         return ...
14     elif d == Direction.E:
15         return ...
16     elif d == Direction.S:
17         return ...
18     elif d == Direction.W:
19         return ...
20
21
```

Fixing the data definition for `Velocity`

```
1  # TODO: Fix the data definition for `Velocity` to use `Direction`
2
3  from typing import NamedTuple
4  Velocity = NamedTuple('Velocity', [('speed', float),
5                                     ('dir', int)]) # in range[0,359]
6
7  # interp. a velocity with its speed in m/s and direction
8  # as an angle in degrees with east=0 increasing counterclockwise
9
10 v1 = Velocity(9, 22)
11 v2 = Velocity(3.4, 180)
12
13 # template based on Compound (2 fields)
14 @typecheck
15 def fn_for_velocity(v: Velocity) -> ...:
16     return ...(v.speed, v.dir)
17
18
```

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

Fixing the data definition for `List[Velocity]`

```
In [ ]:   1  from typing import List
          2  # List[Velocity]
          3  # interp. a list of velocities
          4
          5  LOV0 = []
          6  LOV1 = [Velocity(3.1, 41)]
          7  LOV2 = [V1, V2]
          8  LOV3 = [Velocity(5.9, 265), Velocity(3.5, 89), Velocity(7.9, 323)]
          9
         10  @typecheck
         11  # Template based on arbitrary-sized and reference rule
         12  def fn_for_lov(lov: List[Velocity]) -> ...:
         13      # description of the accumulator
         14      acc = ... # type: ...
         15      for v in lov:
         16          acc = ...(fn_for_velocity(v), acc)
         17      return ...(acc)
         18
         19
```

iClicker question: Where to apply the reference rule?

Target one of the places we'll need to change in order to apply the reference rule in the code above. [Set iClicker question type to "Target".]

▶ ℹ️ Solution (For later. Don't peek if you want to learn 😊)