

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

CPSC 103 - Systematic Program Design

Module 05 Day 1

Rik Blok, with thanks to Ian Mitchell and Giulia Toti

Make-up Monday

This Thursday, October 12 (in two days):

- Your THURSDAY classes will be canceled. Instead, your MONDAY classes will take place at their regularly scheduled time and location on Thursday October 12.
 - So **this** lecture will be canceled this Thursday.
 - Thursday tutorial classes will also be canceled this Thursday.
 - Students in Thursday sections (T1G, T1H, T1J, T1K, T1M, and T1P: check [Thursday tutorial replacements for Make-up Monday](#) Canvas announcement for alternate tutorials offered this week.
-

Reminders

- **Tonight:** Module 4 (Compound): Worksheet
- this Wed-Fri: Module 5 Tutorial Attendance
- Wed: Module 4 (Compound): Code Review
- Wed: Module 2 (HtDF): Tutorial Resubmission (optional)
- Wed: Module 4 (Compound): Tutorial Submission
- Fri: [Project Team Registration](#) (only required if you are doing the project with a partner)

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

Data types

So far:

- Simple atomic
- Interval
- Enumeration
- Optional
- Compound data

Now:

- Arbitrary-sized or *lists* (Module 5)
-

Module learning goals

By the end of this module, you will be able to:

- Understand and use Python's List type.
 - Identify problem domain information of arbitrary size that should be represented as arbitrary-sized data.
 - Use the HtDD and Data Driven Templates recipes with arbitrary-sized data.
 - Design functions that take in and/or return lists.
-

Arbitrary-sized data or List

A `List` is for information that is well represented as a group of elements of the same type:

- a list of names, e.g., `["Ian", "Rik", "Taryn"]`
- a list of integers, e.g., `[1, 1, 2, 3, 5, 8, 13]`
- a list of compound data, e.g., `[velocity(9,22), velocity(3.4,180), velocity(7.6,270)]`

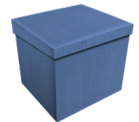
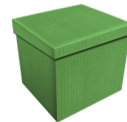
Arbitrary-sized doesn't necessarily mean large

It just means we want flexibility in the number of items we store.

Variables of different types

Simple atomic, interval, enumeration, optional

- One box, one value
- Different kinds of boxes depending on data being stored (e.g., `str` vs. `bool` vs. `Enumeration` vs. `Optional`)



A function can pass back any of these "boxes" as a return type, or any of the boxes below.

Compound

- One box can contain several other boxes, including different kinds of boxes
- Every instance of a container holds the same number and kinds of boxes inside





List

- A container that can stretch to fit a stack of boxes, as long as they're all boxes of the same size and kind




iClicker questions: Pick the best data type



Consider the problem of representing real estate information. Pick the best data type for each of the following kinds of information:

1. A property (including building type, square footage, year built, etc.)

▶  Hint (for your review after class)

▶ Next

(A) Interval, (B) Enumeration, (C) Optional, (D) Compound, (E) List



Warning: Don't mix data types in a list

A Python list can contain different data types at the same time ("mixing data types").

- Mixing data types in a list is prone to error, so generally speaking it is not recommended.
- Example:

```
MyProperty = [BuildingType.APARTMENT, 902.0, "1989"]
```

Item in list	Value
MyProperty[0]	BuildingType.APARTMENT
MyProperty[1]	902.0

Item in list	Value
MyProperty[2]	"1989"

In CPSC 103:

- We plan to loop over items in a list and act on them, expecting that they will all be the same kind of data.
- For consistency and good practices, we will require that **all items in a list must be of the same type**, even though Python will not enforce that restriction.

► Jump to...

List[str] Data Definition

```
In [ ]: 1 from typing import List
2
3 # List[str]
4 # interp. a list of strings
5 LOS0 = []
6 LOS1 = ["hello", "goodbye", "Beatles"]
7
8 @typecheck
9 # template based on arbitrary-sized
10 def fn_for_los(los: List[str]) -> ...:
11     # description of the accumulator
12     acc = ... # type: ...
13
14     for s in los:
15         acc = ... (s, acc)
16
17     return ... (acc)
18
```

Applying the HtDD recipe to lists

Data definition

- We usually don't give the type a name because it's already self-explanatory

```
ListOfStrings = List[str]
```

would be redundant. Instead we just add a comment that our new data type is a

```
# List[str]
```

Interpretation

- Can add more information if you know more about the data being stored.
- But not required 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_los(los: List[str]) -> ...:
```

- we will prepare template with "accumulator" variable, `acc`
- helps us keeps track of something interesting we're looking at in the list:

```
# 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:

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

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

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

Designing a function with accumulators

A `for` loop enables a particular set of statements to be executed repeatedly **for all elements in a list**.

- The set of statements is identified by indentation (we previously used indentation to identify the set of statements in a function, or the set of statements in the answer of an `if/elif/else` statement).

```
for var_name in the_list:
    statement_1_in_loop_body()
    statement_2_in_loop_body()
    :
    statement_k_in_loop_body()

statement_not_in_loop_body()
```

- `var_name` refers to the current item of `the_list` being processed.
- `var_name` should be a new variable name that clearly identifies it as an item in the list. E.g. `dt` for `List[DataType]`.
- When designing functions, your accumulator's description should have the words "in the list so far" in them. That's what accumulators do: keep track of information about what we've seen **in the list so far**!

Example

Let's design a function to count how many strings in a list are "UBC". The stub and some examples have already been completed. Do we need any more? What's the next step?

► [Jump to...](#)

```
In [ ]: 1 @typecheck
2 def count_UBCs(unis: List[str]) -> int:
3     """
4     return the number of times that "UBC" appears in unis.
5     """
6     return -1 # stub
7
8 start_testing()
9
10 # Here are some examples. Do we need any more?
11 expect(count_UBCs(["UW"]), 0)
12 expect(count_UBCs(["UBC"]), 1)
13 expect(count_UBCs(["McGill", "SFU", "UBC", "MIT", "Harvard"]), 1)
14 expect(count_UBCs(["McGill", "SFU", "MIT", "Harvard"]), 0)
15 expect(count_UBCs(["UBC", "McGill", "SFU", "UBC", "MIT", "Harvard", "UBC"]), 3)
16
17 summary()
18
```

Code tracing

Let's trace the code in [Python Tutor](#). (Don't peek until you have finished HtDF for `count_UBCs()` already.)

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

Clicker question: Main loop



Below, we've mostly designed a function to determine if a list of strings contains "UBC". (You may assume a full set of examples is provided.) Notice that our accumulator is the Boolean variable `found`. Just the code implementation and testing remain to be completed.

```
@typecheck
def contains_UBC(unis: List[str]) -> bool:
    """
    Return True if unis includes "UBC", otherwise return False.
    """
    # return True # stub
    # template from List[str]
    # found is True if "UBC" has been found in the list so far, otherwise False.
    found = False # type: bool

    # "main loop" goes here!

    return found
```

Which of the following `for` statements will correctly implement the "main loop"? Select all that apply. [Set iClicker question type to "Multiple Answer".]

(A)

```
for u in unis:
    found = (u == "UBC")
```

(B)

```
for u in unis:
    found = found or (u == "UBC")
```


(C)

(D)

```

for u in unis:
    if u == "UBC":
        found = True
    else:
        found = False

```

►  Hint (for your review after class)

How does return work in a loop?

Recall, `return (expression)` evaluates `(expression)`, stops the currently running function (no matter what was going on), and returns from the function with the value it got from `(expression)`.

(Note that we can only use `return` inside a function!)

So, in a loop, it ends the loop right away!

Sometimes, this is a desirable behavior. For example, let's see how we could use `return` to design a function to determine if the list contains "UBC", without using an accumulator.

Clicker question: Early returns



We've mostly designed a function to determine if a list of strings contains "UBC". (You may assume a full set of examples is provided.) Just the code implementation and testing remain to be completed.

```

@typecheck
def contains_UBC(unis: List[str]) -> bool:
    """
    Return True if unis includes "UBC", otherwise return False.
    """
    # return True # stub
    # template from List[str]

    # "main loop" goes here!

```

Which of the following `for` statements will correctly implement the "main loop"? Select all that apply. [Set iClicker question type to "Multiple Answer".]

(A)

```

for u in unis:
    if u == "UBC":
        return True
    return False

```

(B)

```

for u in unis:
    if u == "UBC":
        return True
    return False

```

(C)

```

for u in unis:
    if u == "UBC":
        return True
    return False

```

(D)

```

for u in unis:
    if u == "UBC":
        return True
    else:
        return False

```

►  Hint (for your review after class)

Accumulator vs. early return


As we've seen, for some tasks we can write code that implements an "early return" to process a list (e.g., "stop as soon as I find what I'm searching for"). Then we don't need to use an accumulator.

But you are always welcome to design your function to use an accumulator, as per the template. You may see some examples of "early returns" in code you see, but you're not required to use them in your code.

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`. Some examples are provided.

```
In [ ]: 1 # TODO: Redesign the body of contains_UBC to call count_UBCs
        2
        3
        4
        5 start_testing()
        6
        7 expect(contains_UBC([]), False)
        8 expect(contains_UBC([""]), False)
        9 expect(contains_UBC(["UW"]), False)
       10 expect(contains_UBC(["UBC"]), True)
       11 expect(contains_UBC(["McGill", "SFU", "UBC", "MIT", "Harvard"]), True)
       12 expect(contains_UBC(["McGill", "SFU", "MIT", "Harvard"]), False)
       13 expect(contains_UBC(["UBC", "McGill", "SFU", "UBC", "MIT", "Harvard", "UBC"]), True)
       14
       15 summary()
       16
       17
```

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

Refactoring

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](#) and use it as you're solving problems. The same sheet will be provided with your exams.

```
In [ ]: 1 # TODO: Design a data definition for a list of integers
        2
        3
```

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