

```
In [ ]: from cs103 import *  
# Recall, we called help(function_name) to get help about a function.  
# Can also call help with a 'library' (in quotes) to get info about it:  
help('cs103')
```

CPSC 103 - Systematic Program Design

Module 03 Day 1

Ian Mitchell, with thanks to Rik Blok and Giulia Toti

Reminders

- this Wed-Fri: Module 3 Tutorial Attendance
- Wed: Module 2 (HtDF): Code Review
- Wed: Module 2 (HtDF): Tutorial Submission
- Mon: Module 4: Pre-Lecture Assignment
- Mon: Module 3 (HtDD): Worksheet

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

Your progress so far

Module 1:

- You learned the basics of Python syntax and how to use Jupyter notebooks to run Python code
- You gained an understanding of variables and functions


Module 2:

- You learned the How to Design Functions (HtDF) recipe, which allows you to write clear and well-structured functions
-

Module learning goals

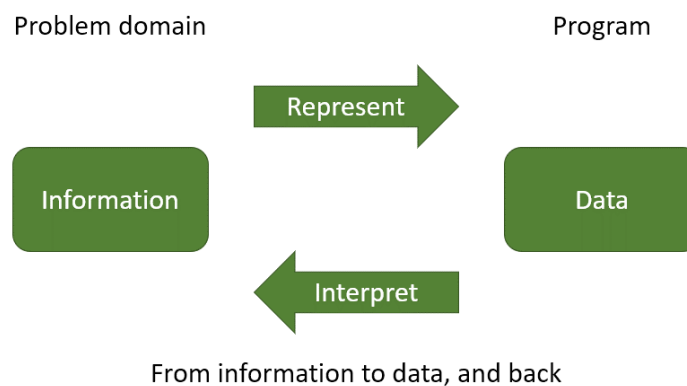
By the end of this module, you will be able to:

- Use the How to Design Data (HtDD) recipe to design data definitions.
- Identify problem domain information that should be represented as simple atomic data, intervals, enumerations, and optionals.
- Use the Data Driven Templates recipe to generate templates for functions operating on data of a user-defined type.
- Use the How to Design Functions (HtDF) recipe to design functions operating on data of a user-defined type.

 Templates - If you felt that we did not properly motivate or explain step 3 "write the template" of the HtDF recipe during module 2, you are absolutely right! - The structure of the template for a function depends on the type of the input data; in other words, the type of the function's argument. In module 2 we only used very simple types of data -- such as `int`, `float`, `bool`, or `str` -- and they all had the simple template `return ...(x)`. - In this module we will introduce several additional types of data and learn how to construct a template for each one. - Modules 4 and 5 will each then introduce a more complicated type of data and the corresponding templates.

Modeling information

- We need to connect the data from our program with information from our problem (and vice versa)
- The HtDD recipe helps us to design and document a representation of the information of the problem as data in the program, and an interpretation of the data in the program as information in the problem.
- Then the HtDF recipe helps us design function(s) which can solve the problem using the data.



iClicker question: Information vs. data



Our program is storing the following data in a variable: `100` . How could this data be interpreted as information in a relevant problem domain?

- A. 100 students in a course
- B. The width of a 100×100 pixel image
- C. \$100 in a bank account
- D. All of the above
- E. None of the above

Information vs. data

- The problem domain guides the *representation* of information as data, and the *interpretation* of data as information.
 - Data without domain knowledge is meaningless!
 - The programmer's job is to bridge the divide so the user can work with *information* and the program can work with *data*.
-

Primitive vs. non-primitive data

Primitive Python data

Data provided by Python, without any meaning attached. They are also called *atomic non-distinct*.

Examples: `int` , `float` , `str` , `bool`

Glossary

- *Primitive*: Built into Python
- *Atomic*: Can't be broken down into smaller pieces
- *Distinct*: A specific value (e.g., `False` , `-3` , or `'hi'`)
- *Non-distinct*: Can take on more than one value (e.g., `bool`)

Non-primitive data

Data we create using a combination of primitive data.

Examples: `age` , `height` , `name` , `grade` , ...

Non-primitive data imparts some information from the problem into Python data, enriching it with some meaning.


How to Design Data recipe (HtDD)

The HtDD recipe consists of the following steps:

1. **Definition:** a line that tells Python the name of the new type (it is equivalent to the function signature from HtDF).
2. **Interpretation:** a comment that describes what the new data type represents (it is equivalent to the purpose comment from HtDF).
3. **Examples:** show how to form data of this type, usually giving special cases.
4. **Template:** a one-parameter function that shows how a function acting on this type of data should operate.

 Q: How to Design Data? A: DIET!

► DIET stands for...

 Don't confuse the HtDD recipe with the HtDF recipe! - HtD**D** and HtD**F** recipes share some similarities; for example, they sound very similar when the acronyms are said fast. - But they serve different purposes: - HtD**F** (SETIT): Given some data, how do you design a **function** to work with it? - HtD**D** (DIET): Given some information to store, how do you **represent** it as data? Given some stored data, how do you **interpret** it as information? - Generally speaking, you cannot start on HtDF until you (or somebody else) has completed the corresponding HtDD. - You'll need to keep them straight and use them correctly.

HtDD naming conventions

1. **Definition** names use UpperCamelCase [\[Wikipedia\]](#). E.g.,

```
MovieTitle = ...
```

2. **Interpretation** starts with `# interp.` .E.g.,

```
# interp. Stores the name of a movie.
```

3. **Example** names are in ALL_CAPS [\[Wikipedia\]](#). They often use an abbreviation or acronym for the data type name, followed by a number or another meaningful mnemonic. E.g.,

```
MT1 = ...
```

```
MT2 = ...
```


```
MT_FAVOURITE = ...
```

4. **Template** function names start with `fn_for_`, then the type name in `snake_case` [\[Wikipedia\]](#). Parameter names are in lowercase (or if necessary `snake_case`), often using an abbreviation or acronym for the data type name. E.g.,

```
def fn_for_movie_title(mt: MovieTitle) -> ...:
```

Aside: Naming style for variables and functions

Reminder that in CPSC 103 we use `snake_case` for all variable, function, and parameter names. See the course [Style Guide](#).

 Benefits of following a naming convention: - Provides additional information to the human reader about the use of an identifier. - Reduces ambiguity for the human reader. - Promotes code sharing and re-use by a human reader. ****Remember:**** The primary target audience for the code you write is a human reader, not Python / Jupyter. You will be expected to follow our naming conventions throughout CPSC 103.

Data types

Our data definitions will be composed from data types provided by Python (e.g., `int`, `float`, `str`, `bool`).

There are several types of data, and there is *no firm rule* for when to use a particular one... it should just fit your problem. For example, depending on the problem, we might represent a temperature as an `int` or a `float` or something else. Through practice, we will learn what factors to consider when choosing an appropriate data type.

Now (Module 3):

- Simple atomic data
- Interval
- Enumeration
- Optional

Later:

- Compound data (Module 4)
 - Arbitrary-sized (Module 5)
-

Simple atomic data

When the information to be represented is itself atomic in form. Usually these are just the primitive data with a better name and description.

```
Temperature = float                                # 1. Definition
# interp. the air temperature in degrees Celsius  # 2. Interpretation

T1 = 0.0                                           # 3. Examples
T2 = -24.5

@typecheck
# template based on Atomic Non-Distinct           # 4. Template
def fn_for_temperature(t: Temperature) -> ...:
    return ...(t)
```



iClicker question: Simple atomic

Which of the following are examples of information that would be best represented with **simple atomic** data types? Select **ALL** that apply. [iClicker: Set to "Multiple Answer" type.]

- A. The temperature of liquid water at standard atmospheric pressure
- B. Allergies that a patient has
- C. The name of a book
- D. A blood type, such as A, B, AB, or O
- E. A phone number

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

Interval

When the information to be represented is numbers within a certain, meaningful range.

```
Time = int # in range[0, 86400)           # 1. Definition
# interp. seconds since midnight         # 2. Interpretation

T_MIDNIGHT = 0                           # 3. Examples
T_ONE_AM = 3600
T_NOON = 43200
T_END_OF_DAY = 86399

@typecheck
# Template based on Atomic Non-Distinct # 4. Template
def fn_for_time(t: Time) -> ...:
    return ...(t)
```

- Range is just a comment for the programmer, not enforced by Python
- Square and round bracket notation borrowed from math: `[` or `]` means the endpoint is *included* and `(` or `)` means it is *excluded*
- Use `...` to indicate no limit, e.g., `WaterDepth = float # in range (... , 0]`



iClicker question: Interval

Which of the following are examples of information that would be best represented with **interval** data types? Select **ALL** that apply.

- A. The name of someone's sibling
- B. A percentage score on a test
- C. A temperature in Celsius
- D. Whether a user is logged in
- E. The wavelength of a visible photon

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

Enumeration

When the information to be represented consists of a fixed number of distinct values.

```
from enum import Enum          # need to import Enum data type from
                                library
```

```
Rock = Enum('Rock', ['ig', 'me', 'se'])
# interp. a rock is either igneous ('ig'), metamorphic ('me'), or
# sedimentary ('se')
```

```
# examples are redundant for enumerations
```

```
@typecheck
# Template based on Enumeration (3 cases)
def fn_for_rock(r: Rock) -> ...:
    if r == Rock.ig:
        return ...
    elif r == Rock.me:
        return ...
    elif r == Rock.se:
        return ...
```

- Python treats the cases in the `Enum` definition as allowed distinct values, and uses the "dot notation" instead of strings to represent.
- Advantage over strings: Restricts allowed values

```
my_rock = Rock.ig          # distinct Enum values allowed (good!)
my_rock = "kryptonite"     # any string allowed by Python (bad!)
my_rock = Rock.kryptonite  # will produce error (good!)
```


- Note that `# examples are redundant for enumerations`, so just write that at step 3.
 - One branch per option in the template, all separated with `elif` (don't use `else` to catch other cases)
-



iClicker question: Elif or else?

For the example above, which of the following **enumeration** template function bodies follow proper style? Select **ALL** that apply.

```
### (A)          ### (B) ``python### (C) ``python</div>
``python if r == if r == Rock.ig:   if r == Rock.ig:
Rock.ig: return ... return ... elif r ==return ... elif r ==
elif r ==         Rock.me: return  Rock.me: return
Rock.me: return   ... else: return ...   ... elif r ==
... elif r ==     ``                    Rock.se: return ...
Rock.se: return ...                ``
else: return ... ``
```

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



iClicker question: Enumeration

Which of the following are examples of information that would be best represented with **enumeration** data types? Select **ALL** that apply.

- A. An individual's emergency contact
- B. The day of the week
- C. How much money there is in a bank account
- D. A music genre played by a streaming channel
- E. The number of pages in a book

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

Optional

When the information to be represented is well-represented by another form of data (often simple atomic or interval) except for one (and only one) special case.

```
from typing import Optional  # need to import Optional data type from library
```

```
Countdown = Optional[int] # in range[0, 10]  
# interp. a countdown that has not started yet (None),  
# or is counting down from 10 to 0
```

```
C0 = None  
C1 = 10  
C2 = 7  
C3 = 0
```

```
@typecheck  
# Template based on Optional  
def fn_for_countdown(c: Countdown) -> ...:  
    if c == None:  
        return ...  
    else:  
        return ...(c)
```



iClicker question: Optional

Which of the following are examples of information that would be best represented with **optional** data types? Select **ALL** that apply.

- A. A person's job title
- B. A season of the year
- C. A user's middle name
- D. The age of a voter in a BC election
- E. A description of the weather

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

Templates

Data template

This is a one-parameter function that shows how a function that takes this data as input should operate.

How to use with function template

When writing a function with the HtDF recipe, in Step 3 "Template", use the template from the data definition instead of writing your own:

- Comment out the body of the stub, as usual
- Then **copy the body of the template from the data definition to the function**
 - If there is no data definition, just copy all parameters (as we did in HtDF for atomic non-distinct data)

References: See the [How to Design Data](#) and [Data Driven Templates](#) pages on Canvas

Practice: "Standing" problem

Problem: Design a function that takes a student's standing ([SD](#) for "standing deferred", [EX](#) for "exempted", and [W](#) for "withdrew") and determines whether the student is still working on the course where they earned that standing.

To do this, we first design a data definition for a standing.




iClicker question: Data definition

How do we represent "Standing" in a way that is understandable and meaningful in Python?

- A. Simple atomic
 - B. Interval
 - C. Enumeration
 - D. Optional
 - E. Something else
-

"Standing" solution

Problem: (from above) Design a function that takes a student's standing ([SD](#) for "standing deferred", [EX](#) for "exempted", and [W](#) for "withdrew") and determines whether the student is still working on the course where they earned that standing.

 Importing from libraries You don't need to memorize the library for the data definition: Just look it up on the [exam reference sheet](https://canvas.ubc.ca/courses/123409/files/27906280/download?download_frd=1). However, to make things easier right now: - Simple atomic - doesn't require a library - Interval - doesn't require a library - Enumeration - `from enum import Enum` - Optional - `from typing import Optional`

```
In [ ]: # Standing = ... # TODO!
```

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

Using with our HtDE recipe

Now we can design – using the HtDE recipe – the function that takes a standing and "determines whether the student is still working on the course where they earned that standing."

Notice that the "Template" step in the HtDF recipe changes from **writing** a template to instead **copying** a template.

```
In [ ]: @typecheck
def still_working(s: Standing) -> ...:
    """
    ...
    """
    return 0 # INCORRECT stub

start_testing()

expect(still_working(...), ...)

summary()
```

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

Re-using our "Standing" data definition

A single data definition is generally used for many different functions in a program, so we will design a second function that uses `Standing` here.

ⓘ Ratio of data definitions to functions In CPSC 103 we often only have time to design one example function for a given data definition in a class, tutorial, assignment, or exam; however, you should not interpret that 1:1 ratio as representative of most real problem solutions. In most real problems, we will design multiple functions to perform multiple operations on a single given data type.

Problem: Design a function that takes a standing (as above) and returns an English explanation of what the standing means.

We already have the data definition, which guides our function design. Indeed, the designed function is very similar to the previous one. Finding where it's *different* may tell you a lot about why examples and templates are useful!

```
In [ ]: @typecheck
def describe_standing(s: Standing) -> ...:
    """
    returns an English description of a Standing s
    """
    return 0 # INCORRECT stub

start_testing()

# We've gone ahead and filled in the test cases
# already to help move us along a bit!
# The HtDD recipe tells us we should have one
# test for every value in the Standing enumeration!

expect(describe_standing(Standing.SD),
       "Standing Deferred: awaiting completion of some outstanding requirement")
expect(describe_standing(Standing.EX),
       "Exempted: does not need to take the course even if it is normally required")
expect(describe_standing(Standing.W),
       "Withdrew: withdrew from the course after the add/drop deadline")

summary()
```

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

Well done! Now, write a call to `describe_standing` :

In []: *# Call describe_standing*