

```
In [ ]: 1 from cs103 import *
        2 # Recall, we called help(function_name) to get help about a function.
        3 # Can also call help with a 'library' (in quotes) to get info about it:
        4 help('cs103')
        5
        6
```

CPSC 103 - Systematic Program Design

Module 03 Day 1

Rik Blok, with thanks to Ian Mitchell 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
-

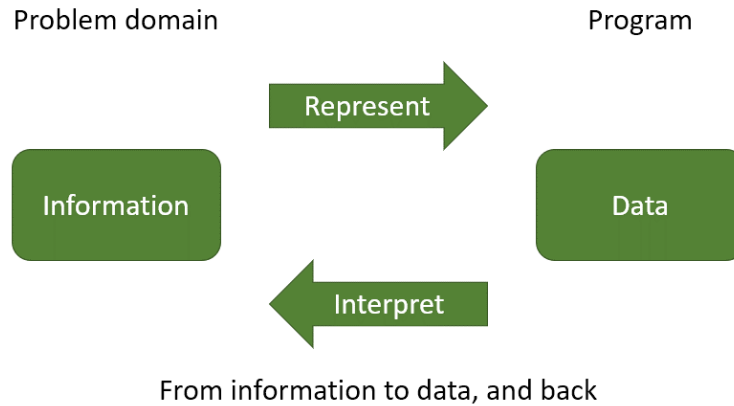
Module learning goals

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.

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 represent the information of the problem as data in your program, using the best format
- Then the HtDF can solve the problem using this 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
- 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**: the line that tells Python the name of the new type, it is like a signature from the HtDF.
2. **Interpretation**: describes what the new data type represents, it is like the purpose from the HtDF.
3. **Examples**: they show how to form data of this type, usually giving special cases.
4. **Template**: this is a one-parameter function that shows how a function acting on this data should operate.

Q: How to Design Data? A: DIET!

► DIET stands for...

Reference: Design recipes

The mnemonics DIET & SETIT may help but you don't need to memorize these recipes. You can find them on our Canvas home page.

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.

Don't confuse HtDD and HtDF recipes

- HtDD and HtDF recipes share some similarities
- But they serve different purposes:
 - HtDD (DIET) = given some information to store, how do you **represent** it as data?

- **HtDF (SETIT)** = given some data, how do you design a **function** to work with it?

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. **Examples** are in ALL_CAPS [\[Wikipedia\]](#) and often use abbreviated type name, followed by a number. E.g.,

```
MT1 = ...  
MT2 = ...  
MT_FAVOURITE = ...
```

4. **Template** function names start with `fn_for_`, then the type name in `snake_case` [\[Wikipedia\]](#).

Parameter names in lowercase, usually abbreviation of data type name. E.g.,

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

5. **(Variables and functions)** - for consistency, let's also use `snake_case` for variables and functions and parameters, moving forward)

See also the course [Style Guide](#).



Benefits of following a naming convention

- Provides additional information about the use of an identifier
- Reduces ambiguity
- Promotes code sharing and re-use

You will be expected to follow our naming conventions for data definitions.

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 to choose an appropriate data type.

Now:

- 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: `[]` means the endpoint is *included* and `()` 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 cases in definition as allowed distinct values, using "dot notation" instead of strings
- 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
- One branch per option in template, all separated with elif



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)


```
if r == Rock.ig:
    return ...
elif r == Rock.me:
    return ...
elif r == Rock.se:
    return ...
else:
    return ...
```

(B)

```
if r == Rock.ig:
    return ...
elif r == Rock.me:
    return ...
else:
    return ...
```

(C)

```
if r == Rock.ig:
    return ...
elif r == Rock.me:
    return ...
elif r == Rock.se:
    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 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 operating on this data should operate.

How to use with function template

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

- Comment out the body of the stub, as usual
- Then **copy the body of 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

"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'll need a data definition for a standing, first.



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: 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! Will be provided on the exam reference sheet.

- Simple atomic - doesn't require a library
- Interval - doesn't require a library
- Enumeration - from `enum import Enum`
- Optional - from `typing import Optional`

```
In [ ]: 1 # Standing = ... # TODO!
        2
        3
```


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

Using with our HtDF recipe

Now we can design – using the HtDF 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 [ ]: 1 @typecheck
        2 def still_working(s: Standing) -> ...:
        3     ....
        4     ...
        5     ....
        6     return 0 # INCORRECT stub
        7
        8
        9 start_testing()
       10
       11 expect(still_working(...), ...)
       12
       13 summary()
       14
       15
```

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

Re-using our "Standing" data definition

A single data definition usually gets used for many different functions in your program, but we often only have time for one in class, tutorial, and assignments. Let's do a second design here!

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

```
In [ ]: 1 @typecheck
        2 def describe_standing(s: Standing) -> ...:
        3     """
        4     returns an English description of a Standing s
        5     """
        6     return 0 # INCORRECT stub
        7
        8
        9 start_testing()
       10
       11 # we've gone ahead and filled in the test cases
       12 # already to help move us along a bit!
       13 # The HtDD recipe tells us we should have one
       14 # test for every value in the Standing enumeration!
       15
       16 expect(describe_standing(Standing.SD),
       17        "Standing Deferred: awaiting completion of some outstanding requirement")
       18 expect(describe_standing(Standing.EX),
       19        "Exempted: does not need to take the course even if it is normally required")
       20 expect(describe_standing(Standing.W),
       21        "Withdrew: withdrew from the course after the add/drop deadline")
       22
       23 summary()
       24
       25
```

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

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

```
In [ ]: 1 # Call describe_standing
        2
        3
```