CPSC 103: Module 2 Day 1

Announcements

- **Reminder**: Karina has a list of announcements and to dos at http://www.cs.ubc.ca/~mochetti/CPSC103.html
- The Canvas sidebar also has a list of upcoming due dates
- Finish working on the module 1 tutorial and code review
- Start on the module 2 worksheet, tutorial, and code review

How does this relate to last week?

- First 1.5 weeks focused on having you learn Python basics and exploring it
- What does design mean?
  - Clearly identifying the problem
  - Clearer about how to go about solving it
  - Hopefully this will help give some context to the Python we've been doing so far

Why HtDF?

- A basic framework to follow
- Gives us some concrete steps to follow to make the process of writing code easier.
- These are steps that expert programmers may not do explicitly but they definitely think about the same things as they are writing code
- Steps are not specific to Python

A Really Summarized Version of HtDF

1. Write the stub, including signature, purpose, and typecheck annotation.

2. Write examples

3. Write or copy the template

4. Code the function body

5. Test and debug until correct

HtDF Step 1: @typecheck

```
@typecheck
```

What is @typecheck for?

- Python will check for restrictions as you use the functions or have other pieces of code use the function you just wrote.
- Python isn't especially smart. You need to specify each function you want it to typecheck.
  - **Put @typecheck before each function**

@typecheck Example

```
1 # This is just an example and not the
2 # full HtDF design!!
3 def repeated_str(s):
4     return s*2
5
6 repeated_str("cat")
```

```
'catcat'
```

What stops someone from calling our function with an int?

```
repeat_str(2)
```

@typecheck Example

```
1 # This is just an example and not the
2 # full HtDF design!!
3 def repeated_str(s):
4     return s*2
5
6 repeated_str(2)
```

```
4
```

Is this something we want to allow?

How can we prevent it?

@typecheck Example

```
1  from cs103 import *
2
3  # This is just an example and not the
4  # full HtDF design!!
5  @typecheck
6  def repeated_str(s: str) -> str:
7      return s*2
```

```
1  from cs103 import *
2
3  # This is just an example and not the
4  # full HtDF design!!
5  @typecheck
6  def repeated_str(s: str) -> str:
7      return s*2
8
9  repeated_str('cat')
```

```
'catcat'
```

@typecheck Example

```
1  from cs103 import *
2
3  @typecheck
4  def repeat_str(word:str) -> str:
5      return word*2
6
7  repeat_str(2)
```

```
---------------------------------------------------------------------------
TypecheckError

while checking parameter word: 2 is an int, not a str

  File "<ipython-input-4-356d444cb92b>", line 7, in <module>
    repeat_str(2)

as expected in

  @typecheck
  def repeat_str(word:str) -> str:
      return word*2
```

HTDF Step 1: Signature

```
@typecheck

def foo(lucky_number: int) -> int:
```

This is the signature of the function.

It tells us the name of the function, inputs, and outputs.

HtDF Step 1: What is a purpose?

- It helps other people understand your code.
- Reading the description is often easier than reading code
- Especially helpful when there are lots of functions inside one file and you want to quickly figure out what functions may be relevant to what you want.
- No one wants to read a lot of code at once!
- Sometimes, you will hear the purpose referred to as a docstring

What is a docstring?

- The docstring is what is produced when you call help()

```
1  from cs103 import *
2
3  def foo(lucky_number: int) -> int:
4      """
5      Multiples the lucky_number by 7 to create another lucky number
6      """
7      # This is not the full HtDF design!!
8      return lucky_number * 7
9
10 help(foo)
```

```
Help on function foo in module __main__:

foo(lucky_number: int) -> int
    Multiples the lucky_number by 7 to create another lucky number
```

HtDF Step 1: Stub

```
@typecheck

def foo(lucky_number: int) -> int:

    return 0 # stub
```

The stub **MUST** match the return data type indicated in the signature. It can be any value as long as the type matches.

HtDF Step 2: Examples

- Write at least one example of a call to the function and the result that the function is expected to return.
- You want to determine what it means for your function to behave correctly BEFORE you implement it
- A good rule of thumb:
  - Write a test by calling the function the first reasonable way you imagine.
  - Write at least one more test to show how the function's behaviour changes if you chance the inputs (argument values).
  - Think through if there are any special cases to illustrate.

HtDF Step 2: Examples

- When creating tests for your function, you don't need examples that will break your signature.
  - E.g., If your function accepts a str as a parameter, you don't need to write examples that test what happens when you pass an int into your function.

HtDF Step 2: Examples

```
@typecheck

def foo(lucky_number: int) -> int:

    return 0 # stub

start_testing

expect(foo(7), 49)
expect(foo(-2), -14)
expect(foo(0), 0)

summary()
```

HtDF Step 3: Template

- The template is always based on the **input** types
- This provides a general structure to help with the implementation of your function

HtDF Step 3: Template

```
@typecheck

def foo(lucky_number: int) -> int:

    # return 0 # stub

    return …(lucky_number) # template

start_testing

expect(foo(7), 49)
expect(foo(-2), -14)
expect(foo(0), 0)

summary()
```

HtDF Step 4: Code the function body

- Make use of the template when implementing the function body.
- `return …(lucky_number)` tells us that in our implementation, `lucky_number` is a variable that we might want to use

HtDF Step 4: Code the function body

```
@typecheck

def foo(lucky_number: int) -> int:

    # return 0 # stub

    # return …(lucky_number) # template

    return lucky_number * 7

start_testing

expect(foo(7), 49)
expect(foo(-2), -14)
expect(foo(0), 0)

summary()
```

HtDF Step 5: Run and test your code

- Run your code
- Do all the tests pass?

  If you have done a good job writing tests, then you can fairly assured that your code is working as intended.