```
In [ ]: from cs103 import * # needed (once per notebook) to enable incredible cs103 powers!
```

CPSC 103 - Systematic Program Design

Module 02 (HtDF) Day 1

Ian Mitchell, with thanks to Rik Block and Giulia Toti

Reminders

- Today: Confirm that your iClicker responses have been recorded in Canvas. If not, make sure you have set up your iClicker cloud account properly.
- Wed: Module 1 (Intro): Code Review
- Wed: Module 1 (Intro): Tutorial Submission
- this Wed-Fri: Module 2 Tutorial Attendance
- Mon: Module 3: Pre-Lecture Assignment
- Mon: Module 2 (HtDF): Worksheet

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

Exam practice exercises - by module

After each module: self-assess and practice for exams at https://canvas.ubc.ca/courses/123409/pages/extra-exercises-written

Designed to help you assess your exam preparation. For a more accurate assessment:

- practice on paper, to better simulate exam conditions
- practice without distractions and interruptions
- time yourself
- do not use any notes or other resources that would not be available during the real exam (but you can definitely use the exam reference sheet
- only check the solution when you are done with all the exercises

---

Module learning goals

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

- use the How to Design Functions (HtDF) recipe to design functions that operate on primitive data.
- read a complete function design and identify its different elements.
- evaluate the elements of a function design for clarity, simplicity, and consistency with each other.
- evaluate an entire design for how well it solves the given problem.
- explain the intended purpose of the HtDF recipe's steps in a way that generalizes to other design problems.

---

iClicker question

You want a program that will display the appropriate response in these cases:

- "That is very warm" if `temp` > 30,
- "That is warm" if `temp` is in the range [20,30] (inclusive),
- "That is cool", otherwise.

Which of the following is correct, assuming `temp` has been assigned some numerical value?

(A)

```
out = "That is "
if temp > 30:
    out = out + "very "
elif temp >= 20:
    out = out + "warm"
else:
    out = out + "cool"
out
```

(B)

```
out = "That is "
if temp > 30:
    out = out + "very "
if temp >= 20:
    out = out + "warm"
else:
    out = out + "cool"
out
```

(C) Both

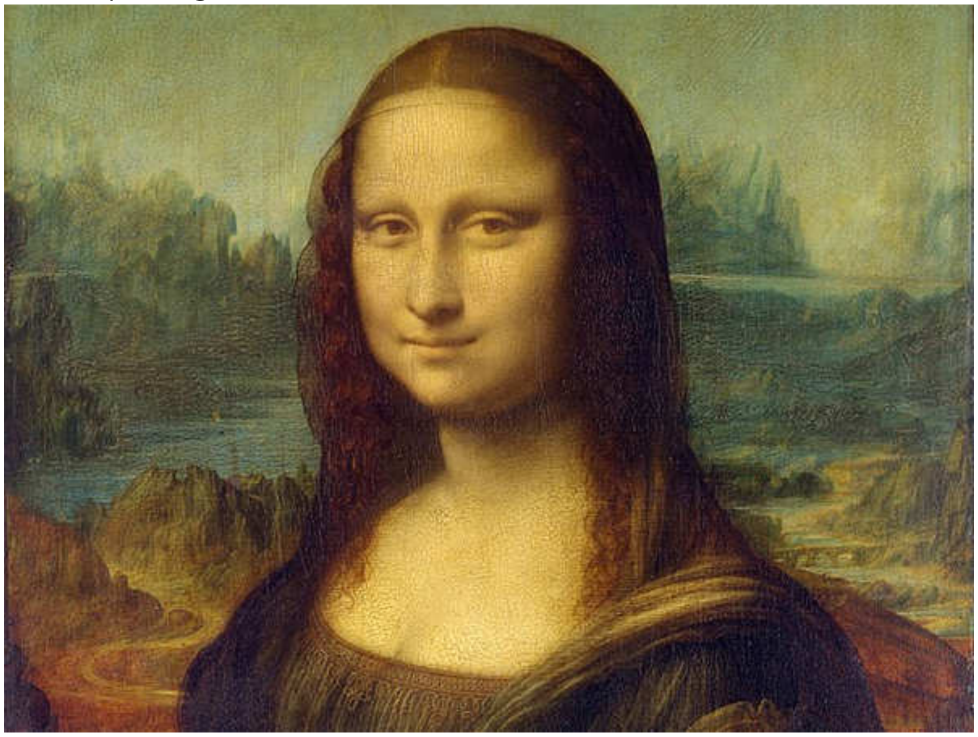
(D) Neither

▸ i Hint (for your review after class)

```
In [ ]:
```

About good programming

From How to Design Programs, Second Edition: "...programming differs from good programming like

> crayon sketches in a diner
>
>
>
> from oil paintings in a museum"
>
> 

After this course...

Maybe you won't be like him (yet!)...

...but you could be like them!

How to Design Functions (HtDF) recipe

The HtDF recipe consists of the following steps:

1. Write the **s**tub, including signature and purpose
2. Define **e**xamples
3. Write the **t**emplate
4. **I**mplement the function body
5. **T**est and debug until correct

See HtDF page on Canvas. For notation and naming conventions that **must** be used in CPSC 103, see the Style Guide on Canvas.

i Q: How do I remember this recipe? A: SET-IT!

▸ SET-IT stands for...

Step 1 Write the <u>S</u>tub, including signature and purpose

Stub - a temporary placeholder for some yet-to-be-developed code

▸ Jump to...

Three substeps to write the stub:

1a. Signature
1b. Purpose
1c. Stub return

Step 1a: Signature

Defines the **input type(s) and output type** for your function. It is also when you give a **name** to your function and to the parameters. Names should be **all lower case**, with underscore (_) separating words.

▸ Jump to...

Example: Design a program that returns the double of a given number.

```
In [ ]:
```

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

Aside: @typecheck

What does `@typecheck` do? Let's find out!

In [ ]:
```
help(typecheck)
```

We will annotate every function we define with `@typecheck`. Let's start with our example function above.

Step 1b: Purpose

The purpose statement describes what the function will return. It is written in triple quotes at the top of a function's body and documents the function.

▸ Jump to...

Example: Design a program that returns the double of a given number.

In [ ]:
```

```

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

Aside: Docstrings

- "Triple quotes" (which can be written as `'''` or `"""`) denote a *multiline string*, a string that can contain linebreaks. Otherwise it is the same as single or double quotes to define a string.
- In this case that multiline string is an expression which happens to be a literal of type `str`. Following Python execution rules, this expression is evaluated and then the result is discarded (because we did not do anything with it, such as assign it to a variable).
- Special Python rule: When a string is placed immediately below a function definition, that string is treated as documentation or a *docstring*. The docstring for function `foo` (assuming that you have defined such a function yourself or imported it from somewhere) can be accessed by the expression `help(foo)`.

Example:

In [ ]:
```
# display documentation for function double
help(double)
```

Note that I could also have written the documentation in a comment (`# ...`). But then `help` would not have displayed it.

In CPSC 103, we will write the purpose statement inside the docstring at the top of every function we define.

Step 1c: Stub return

The stub must return a value of the right type. It doesn't need to be correct; any value of the right type will do. The stub return must also be commented as such, with `# stub` .

▶ Jump to...

Example: Design a program that returns the double of a given number.

In [ ]:

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

We will add a stub return to every function we define.

Step 2 Define Examples

Write *at least* one example of a call to the function and the result that the function is expected to return.

▶ Jump to...

Your examples should appear after your function definition. Don't forget to unindent, otherwise Python will think they are part of the function definition. We will use the following notation (using functions from the `cs103` library) to write our examples:

```
...
    return 0 # stub

start_testing()

expect( ... , ... ) # an example
expect( ... , ... ) # another example
...

summary()
```

Example: Design a program that returns the double of a given number.

In [ ]:

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

We will add test cases to cover all conditions we can think of for every function we define.

Step 3 Write the Template

▸ Jump to...

Comment out the body of the stub and copy the body of the template from the data definition to the function design. We will talk about data definitions (and hence templates specific to those data definitions) starting in Module 3. If there is no data definition (which is true for all of our functions during this module), just copy all parameters.

Example: Design a program that returns the double of a given number.

In [ ]:

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

We will write a function template for every function we define.

Aside: Ellipsis (`...`)

Ellipsis (`...`) can be used as a placeholder for future code. Skipped over when run. Useful for testing.

Example:

In [ ]:
```
def future_function_1():
    # Nonfunctional but will run
    ...

def future_function_2():
    # Will throw error
```

Step 4 Implement the function body

Complete the function body by filling in the template, note that you have a lot of information written already. Remember to comment out the template.

▸ Jump to...

Example: Design a program that returns the double of a given number.

```
In [ ]:
```

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

### ⚠ Why the stub and template? You may wonder why we write the stub and template, but then comment them out shortly thereafter. The reasons have to do with good program design and will become clear later in the course.

Step 5 Test and debug until correct

You should run and correct your function until all tests pass.

▸ Jump to...

i Terminology -[Bug](https://en.wikipedia.org/wiki/Software_bug): A defect or problem that prevents correct operation. -[Debugging](https://en.wikipedia.org/wiki/Debugging): The process of finding and resolving bugs ![module02-First_Computer_Bug,_1945_cropped.jpg](attachment:module02-First_Computer_Bug,_1945_cropped.jpg) The moth found trapped in the Mark II Aiken Relay Calculator while it was being tested in 1947. The operators affixed the moth to the computer log, with the entry: "First actual case of bug being found". (The term "debugging" already existed.)

Example: Design a program that returns the double of a given number.

```
In [ ]:
```

▸ i Sample program with bug for testing (For later. Don't peek if you want to learn 🙂)

In [ ]:
```
# Now that your function is fully designed and tested,
# you can use it
```

Questions from the pre-lecture assignment

1. What exactly is the difference between the stub and the template? What do you put in a stub return vs a template return?

▶ ▶ Answer:

1. What is the benefit of using the HtDF recipe as opposed to just defining functions the way we learned to in Module 1? Do programmers use this recipe while working / will we find use for the recipe outside of this class?

▶ ▶ Answer:

Exercise: `is_palindrome` (similar to Pre-Lecture Assignment)

Problem: Design a function that takes a string and determines whether it is a palindrome or not. (A palindrome is a word or phrase that reads the same backwards and forwards - e.g., "level".)

The first step of our HtDF recipe have already been completed below (with two possible purposes included):

1. Done: Write the **S**tub, including signature and purpose
2. TODO: Define **E**xamples
3. TODO: Write the **T**emplate
4. TODO: **I**mplement the function body
5. TODO: **T**est and debug until correct

Step 1: Write the Stub

iClicker questions

1. Where is the function signature?
   (A) Line 3, **(B)** Line 4, **(C)** Lines 6-12, **(D)** Line 13, **(E)** Lines 16-18

▶ ▶ Next

In [ ]:
```python
# Design is_palindrome function here

@typecheck
def is_palindrome(word: str) -> bool:
    # Should the purpose be...?
    """
    return True if the word is a palindrome, and False otherwise
    """
    # Or...?
    """
    return True if the word meets our requirements
    """
    return True  # stub

# Starting point for any set of tests/examples:
start_testing()
expect(..., ...)
summary()
```

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

Now that `is_palindrome` has been designed and tested, we can use it!

In [ ]:
```python
# Write a call to is_palindrome here
```

iClicker check-in

How are you doing? Any trouble keeping up?

A. 💪 Easy-peasy... you can go faster
B. 👍 Yup, I got this
C. 😕 I might have missed a bit here or there
D. 😟 Hmm, something's not working right
E. 😵 I have no idea what's going on
