

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

CPSC 103 - Systematic Program Design

Module 02 Day 1

Rik Blok, with thanks to Prof. Giulia Toti

Reminders

- Wed: Module 1 (Intro): Code Review
- Wed: Module 1 (Intro): Tutorial
- Mon: Module 2 (HtDF): Worksheet
- Mon: Module 2: Pre-Lecture Assignment

See also the [course calendar](#).

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?

►  Hints:

A. `python out = "That is " if temp > 30: out = out + "very " elif temp >= 20: out = out +`

B. `python out = "That is " if temp > 30: out = out + "very " if temp >= 20: out = out + "warm"`

"warm" else: out = out + "cool" out ``

C. Both

else: out = out + "cool" out ``

D. Neither

In []:

iClicker question



Imagine you want to write a function to compute how many 3.8 Litre cans of paint are needed to paint a wall. Which of the following are good arguments for this function? Select **ALL** that apply.

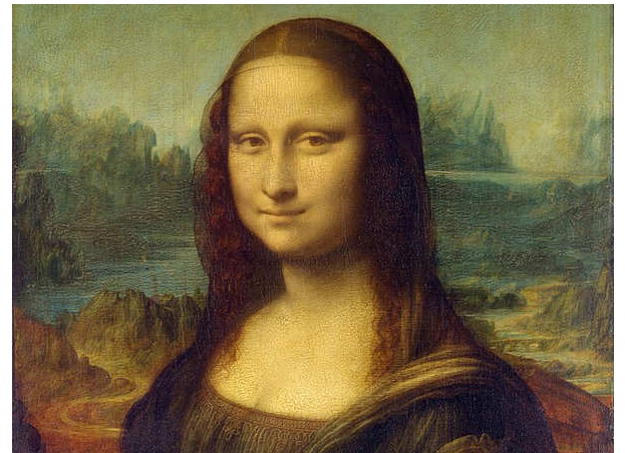
- A. Height of the wall
- B. Width of the wall
- C. Thickness of the wall
- D. Size of a can
- E. Number of cans needed

► Details

About good programming

...programming differs from good programming like crayon sketches in a diner from oil paintings in a museum.

– [How to Design Programs, Second Edition](#)



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 stub, including signature and purpose
2. Define examples
3. Write the template
4. Code the function body
5. Test and debug until correct

Step 1: Write the stub, including signature and purpose

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

- 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.

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

► Details

In []:

Aside: @typecheck

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

In []: `help(typecheck)`

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 that documents the function.

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

► Details

In []: `@typecheck
def double(n: float) -> float:`

Aside: Docstrings

- Triple quotes denote a *multiline string*, a string that can contain linebreaks
- Just a literal string. Expression is evaluated but discarded
- When placed immediately after a function definition, treated as documentation
- Note that `help` (below) would not display purpose if I had written it as a comment instead (`# ...`)

Example:

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

Step 1c: Stub return

The stub must return a value of the right type. It doesn't need to be correct, any value from the right type will do.

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

► Details

In []: `@typecheck
def double(n: float) -> float:
 """
 Returns n doubled
 """`

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.

► Details

```
In [ ]: @typecheck
def double(n: float) -> float:
    """
    Returns n doubled
    """
    return 0 # stub
```

Step 3: Write the template

Comment out the body of the stub and copy the body of template from the data definition to the function design. In case there is no data definition, just copy all parameters.

► Details

```
In [ ]: @typecheck
def double(n: float) -> float:
    """
    Returns n doubled
    """
    return 0 # stub

start_testing()
expect(double(0), 2 * 0)
expect(double(-2.1), 2 * -2.1)
expect(double(10.11), 2 * 10.11)
summary()
```

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: Code 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 the template.

► Details

```
In [ ]: @typecheck
def double(n: float) -> float:
    """
    Returns n doubled
    """
    # return 0 # stub
    return ... (n) # template

start_testing()
expect(double(0), 2 * 0)
expect(double(-2.1), 2 * -2.1)
expect(double(10.11), 2 * 10.11)
summary()
```

Step 5: Test and debug until correct

You should run your functions until all tests pass.

► Details

```
In [ ]: @typecheck
def double(n: float) -> float:
    """
    Returns n doubled
    """
    # return 0 # stub
    # return ... (n) # template
    return n ** 2

start_testing()
expect(double(0), 2 * 0)
expect(double(-2.1), 2 * -2.1)
expect(double(10.11), 2 * 10.11)
summary()
```

```
In [ ]: # Now that your function is fully designed and tested,
# you can use it
double(4.6)
```

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:

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 stub, including signature and purpose
2. TODO: Define examples
3. TODO: Write the template
4. TODO: Code the function body
5. TODO: Test 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 [ ]: # 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()
```




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

```
In [ ]: # 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
