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

CPSC 103 - Systematic Program Design

Module 02 Day 2

Ian Mitchell, with thanks to Rik Blok and Giulia Toti

Reminders

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

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

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.

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

Remember to run and check your program after each step. The program may throw errors or behave incorrectly, but its behaviour should align with your expectations. Don't let bugs accumulate!

Continuing...

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. Done: Define **E**xamples
3. Done: Write the **T**emplate
4. Done: **I**mplement the function body
5. TODO: **T**est and debug until correct

Step 1: Write the Stub

Step 2: Define Examples

Step 3: Write the Template

Step 4: Implement the function body

- Hint: Notice what the slice `[::-1]` does to a string. (Try it! 🙂)
- Compare returning a boolean to using `if else`
- Use `string.lower()` and `string.upper()` to convert case

Step 5: Test and debug until correct

In [11]:
```
# Design is_palindrome function here

@typecheck
def is_palindrome(word: str) -> bool:
    """
    return True if the word is a palindrome, and False otherwise
    """
    # return True  # stub
    # return ...(word) # template
    if word == word[::-1]:
        return True
    else:
        return False

# Starting point for any set of tests/examples:
start_testing()
expect(is_palindrome("racecar"), True)
expect(is_palindrome("Racecar"), True) # neglect case
expect(is_palindrome("motor"), False)
expect(is_palindrome("a"), True)
expect(is_palindrome("Hi mom!"), False)
expect(is_palindrome("race car"), False)
expect(is_palindrome("!level!"), True)
expect(is_palindrome(""), False)

summary()
```

```
Test failed: expected True but got False
    Line 18: expect(is_palindrome("Racecar"), True) # neglect case
Test failed: expected False but got True
    Line 24: expect(is_palindrome(""), False)
6 of 8 tests passed
```

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

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

Exercise: Multiply a number by 4

Problem: Design a function that multiplies a number by 4.

In [ ]:
```
# Design your function here
```

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

Aside: Testing outside of CPSC 103

`expect` is provided by `cs103` library and you're expected 😉 to use it in this course.

To test your own Python code outside of this course, you can use the `assert` keyword instead:

In [7]:
```
assert ...
```

Python: Global vs local variables

Variables are defined when they're first assigned. Two types of variables:

- **Global** - defined outside of any function. Can be accessed anywhere after definition: in functions called later, or outside of functions
- **Local** - defined within a function. Can only be accessed within that function. Lost when program leaves the function

⛔ Don't use global variables in functions! To make the code easier to understand **avoid using global variables in functions**. Maintain the "boundaries" of your functions. The **only** data going into a function is through its arguments, and the only data coming out comes from the `return` statement. Need to use a global variable in a function? Pass it in as an argument! For clarity, use different names for global variables and function parameters.

⛔ Example, using a global variable in a function (bad!)

In [ ]:
```
def step(a: float) -> float:
    return a + step_size

start_testing()

step_size = 1
expect(step(5), 6) # Example 1
# The next statement could be much more complicated,
# hiding the change to this global variable!
step_size = -1
# Global variable in function breaks functional programming paradigm.
# Function output not reliable, depends on more than just inputs.
expect(step(5), 6) # Example 2 appears identical to Example 1, but the test fails

summary()
```

✅ Same example, let's fix it to pass global variable as an argument (good!)

In [ ]:

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

iClicker question

Why is it considered bad style to use global variables in functions? Select **ALL** that apply. [iClicker: Multiple Answer question]

Global variables can...

A. make it difficult to understand the dependencies between different parts of a program. This can make it hard to change code
B. cause functions to be affected by external factors, other than their inputs
C. make it difficult to test a program, as they can lead to unexpected interactions between different parts of the code

iClicker question

What is the output of the program to the right? (Please forgive the improper function design 🙂) [iClicker: Multiple Choice question]

A. `'home to class'`
B. `'home to home'`
C. `'class to home'`
D. `'class to class'`
E. Something else

```python
def left_to_right(left:str, right:str)->str:
    return left + " to " + right

left = "home"
right = "class"
left_to_right(right, left)
```

▶ ℹ️ Hints (for your review after class)

Exercise: Discount

Problem: A store gives 5% off the most expensive product and 10% off the second most expensive product. Design a program that calculates the final purchase price of any two given products.

In [ ]:
```
# An improperly designed function signature is provided here as a starter

def discount(price1, price2):
```

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

Now that our `discount` function has been designed and tested, we can use it!

In [ ]:
```
p1 = 50
p2 = 75
```

---

iClicker question

In the function `discount` defined above, which function parameter receives the value 10 when the code to the right is run?

A. `p1`
B. `p2`
C. `price1`
D. `price2`
E. Two of the above

```python
p1 = 20
p2 = 10
discount(p2, p1)
```

▶ ℹ Hints (for your review after class)

---

iClicker question

The program on the right calculates the total value of a product after adding a 12% tax. Find the design errors of this program. Select **ALL** that apply. [iClicker: Multiple Answer question]

A. Signature	```python @typecheck def AddTax(value: float) -> float: """
B. Purpose	returns the total after adding a 12% tax on value """ return 0 #
C. Stub return	stub #return value # template return value*0.12 + value
D. Examples	start_testing() expect(AddTax(10), 11.2) expect(AddTax(12), 13.44)
E. Template	expect(AddTax(500), 560) summary() ```

▶ ℹ Hints (for your review after class)

CPSC 103 d-tective 😜

Problem: Design a function that determines if a string starts with the letter *d*.

⚠ Is the problem well-defined? As you start working through this problem, you may discover that the problem is slightly ambiguous. Think about how you should deal with vague problems when designing programs.

In [ ]:

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