Welcome back to CS 2100!

Prof. Rasika Bhalerao

Poll: How can I swap the values of `x: int` and `y: int`?

a.

```
temp: int = x
x = y
y = temp
```

b.

```
temp: int = x
temp = y
y = x
```

c.

```
temp: int = y
x = y
x = temp
```

d.

```
temp: int = x
y = temp
x = y
```

Poll: What does this print? `print("I am so excited" + "!" * 3)`

1. I am so excited! I am so excited! I am so excited!
2. I am so excited!I am so excited!I am so excited!
3. I am so excited!!!
4. Nothing – it breaks

Fun with Python: `str` times `int`

We use quotes to represent strings.

Single quotes `'hi'` or double quotes `"hi"`

So... How can we put quotes in a string?

```
"Easiest way to put an apostrophe ' is to use double quotes '"
'Easiest way to put quotes " is to use single quotes "'
```

What if I want both `'` and `"` in the string?

Python documentation that may help

Thanks, but that was very ugly.

F-string

Preferred solution: put the variable directly in the string with `{` brackets `}`

```python
cats: int = 4
print(f"There are {cats} cats in this room.")
```

```
There are 4 cats in this room.
```

Cleaner, reduces opportunities for bugs, impresses your boss...

Float

Normal activity: run this:

```
print(5 / 2)
print(type(5 / 2))
```

```
2.5
<class 'float'>
```

Fun activity: run this:

```
print(4 / 2)
print(type(4 / 2))
```

Formatting floats

Let's figure out how to print `price` with exactly two decimal places.

Starting point:

```
price = 12.345678
print(f'{price}')
```

Evaluate:

```
7 < 2 + 8
```

Evaluate:

```
3 < 4 and 5 < 7
```

Order of operations

- Math happens before comparison operations
- Comparison happens before boolean operations

Conditionals: if / else

Let's fill in this function:

```python
def print_cats(num_cats: int) -> None:
    """Prints the number of cats in this format: 4 cats
    Adds the 's' only if num_cats is plural.

    Raises:
        ValueError: If num_cats is negative.
    """
```

Tip: we can put a conditional expression in one line:

```python
print('yes' if my_decision else 'no')
```

Conditionals: match case statements

Practical when there are many cases

Let's add cases and run this:

```python
name: str = input('Please enter your name: ')
match name:
    case 'SpongeBob':
        print('You are a sponge')
    case _:
        print("I don't know you")
```

- Finds the first case that matches, and only executes that one case
  - or zero cases if none match
- `case _` is a catch-all that matches anything that didn't fit any other cases
  - Not required, but if it is there, it must be the last case

Iteration: for loops over range of numbers

Helpful function: `range()`

```
for i in range(4):
    print(i)

>> 0
   1
   2
   3
```

Can start at a number other than 0:

```
for i in range(2, 5):
    print(i)

>> 2
   3
   4
```

Can count in "steps" larger than 1:

```
for i in range(10, 30, 5):
    print(i)

>> 10
   15
   20
   25
```

What's the silliest way you can count from 1 to 10? Inspiration:

```
for i in range(-30, -130, -10):
    print(int(-(i + 30) * 0.1) + 1)
```

Iteration: for loops over the elements of a collection

The `range()` function returns a collection, which the for loop iterates over.

Can iterate over the elements of a different collection:

```
for character in 'I love cats!':
    print(character.upper())
```

```
I
 
L
O
V
E
 
C
A
T
S
!
```

```python
def sarcasm(phrase: str) -> str:
    """Returns the sarcastic version
    of the provided phrase, where a
    randomly selected half of the
    characters are uppercase, and
    the others are lowercase.

    Parameters
    ----------
    phrase : str
        The phrase to turn sarcastic
    Returns
    -------
    str
        The sarcastic version of
        the phrase
    """
    sarcastic_phrase = ''
    for character in phrase:
      if random() < 0.5:
        sarcastic_phrase +=
            character.upper()
    return sarcastic_phrase
```

Poll: What's wrong? Why doesn't the docstring match the code?

1. It's adding the index of the character, not the character itself
2. It skips adding about half of the letters
3. Sometimes, it doesn't return a string at all
4. It adds extra characters to the string

What if I want the elements *and* the indices?

Use `enumerate()`:

```
for index, word in enumerate(['American Shorthair', 'Balinese', 'Cheetah']):
    print(f'{index}: {word}')

>> 0: American Shorthair
   1: Balinese
   2: Cheetah
```

Taking a step back: we've been using `input(prompt)` and `print()` a lot...

How do we test those?

While testing, we can't rely on the user to:

- type the right thing in `input()`
- verify things that were `print()`ed

Solution: "Mock" the user

`unittest` allows us to pretend to be a person typing stuff in

Source: https://en.meming.world/wiki/Mocking_SpongeBob

Tests that mock user input

```python
import unittest
from unittest.mock import patch, Mock

def concat_three_inputs() -> str:
    """Reads three inputs from the
    user and concatenates them into
    a single string separated by spaces.
    """
    inputs = []
    for _ in range(3):
        user_input = input(
            "Enter something: ")
        inputs.append(user_input)
    return ', '.join(inputs)
```

```python
class TestConcatThreeInputs(unittest.TestCase):
    """Unit tests for concat_three_inputs."""

    @patch('builtins.input', side_effect=[
        'first thing',
        'second thing',
        'third thing'])
    def test_concat_three_inputs(
        self, _: Mock
    ) -> None:
        """Test that concat_three_inputs
        concatenates three user inputs.
        """
        result = concat_three_inputs()
        self.assertEqual(
            result,
            'first thing, second thing, third thing')
```

Notice the `_: Mock` argument to the test function.

Note: If there are not enough inputs specified in the `side_effect` array, the call to `input()` will wait forever (until it times out).

Tests that mock console output

```python
def repeat_three_inputs() -> str:
    """Reads three inputs from the user and prints them."""
    for _ in range(3):
        user_input = input("Enter something: ")
        print(user_input)

class TestRepeatThreeInputs(unittest.TestCase):
    """Unit tests for the concat_three_inputs function."""

    @patch('builtins.input', side_effect=[
        'first thing', 'second thing', 'third thing'])
    @patch('builtins.print')
    def test_repeat_three_inputs(
        self, mock_print: Mock, _: Mock) -> None:
        """Test that repeat_three_inputs correctly
        reads and prints three inputs."""
        repeat_three_inputs()
        expected_calls = [
            unittest.mock.call("first thing"),
            unittest.mock.call("second thing"),
            unittest.mock.call("third thing"),
        ]
        mock_print.assert_has_calls(expected_calls)
```

`@patch` decorators "stack" such that the first decorator is the last argument, and vice versa.

Since we don't use the mock input's argument, we name it `_`.

More things used in assignments...

Refactoring

Refactoring is moving the code around without changing the functionality

Programmers refactor their code to make it more readable, more testable, and easier to modify

We often refactor...

- Code used in multiple places into a single function that gets called multiple times
- Code from a complex function into smaller functions
- (Magic) numbers or string literals into constants

Magic numbers

Magic numbers are unnamed numeric literals in code.

We don't like magic numbers.

We name our literals (except for -1, 0, 1, and 2) to make our code self-documenting.

We name our literals by making them into **constants**: variables named in `UPPER_SNAKE_CASE` that aren't meant to be modified while the programming is running.

Why use named constants?

- Readability

```
SECONDS_PER_MINUTE = 60
MINUTES_PER_HOUR = 60
HOURS_PER_DAY = 24
SECONDS_PER_DAY = SECONDS_PER_MINUTE * MINUTES_PER_HOUR * HOURS_PER_DAY
```

- Safety

```
timer(SECONDS_PER_DAY)
timer(86400)
```

- Maintainability

```
CREDITS_TO_GRADUATE = 128
NUMBER_OF_CAMPUSES = 10
```

Import code

Import modules like `import unittest`

Can also import code from a file that we wrote ourselves: `import my_file`

When a Python file is imported, all of the code inside it is executed. That's why we put our code inside functions -- we don't want the code inside to be executed when it's imported!

Call all the functions in `main()` and add this at the end of the file:

```
if __name__ == '__main__':
    main()
```

Import code: trying it out in lecture

1. Put `print('hello')` in a new file, import it in this file, and run this file (where it is imported)
2. Move it to inside a function and run it
3. Add the `if __name__ == '__main__'` conditional at the end and run it
4. Run this file (where it is imported) to make sure it doesn't print

Poll:

1. What is your main takeaway from today?
2. What would you like to revisit next time?