Announcements & Logistics

- **HW 5** will be released today on GLOW
- **Lab 4** Part 1 due Wed/Thurs 10 pm
  - We will return feedback (including tests not found in `runtests.py`)
- Reminder that Midterm is **Thursday March 14**
  - Two exam slots: 6-7.30 pm, 8-9.30 pm
  - Room: Bronfman auditorium
- Midterm review Monday March 11 evening 7-9 pm in Bronfman Auditorium
- How to study: review lectures
  - Practice past HW and labs (using pencil and paper)
  - Additional [POGIL](#) worksheets posted on course website (resources)

Do You Have Any Questions?
Last Time

• Wrap upped up OSCAR example (more for loops and nested lists)
• Introduced list comprehensions
  • Short-hand expressions for common looping patterns
  • Anything you can do with a list comprehension can be done using the techniques we've discussed so far; very "Pythonic" idiom
Today's Plan

• New iteration statement: the **while** loop
• Discuss the *mutability* of different data types and the implications
When you don't know when to stop (ahead of time):

While Loop
Story so far: **for** loops

- Python **for** loops are used to iterate over a **fixed sequence**
  - No need to know the sequence's length ahead of time
- Interpretation of for loops in Python:
  ```python
  for thing in things:
      (do something with thing)
  ```
- Other programming languages (like Java) have **for** loops that require you to explicitly specify the length of the sequence or a stopping condition
- Thus Python for loops are sometimes called “**for each**” loops
- **Takeaway:** For loops in Python are meant to iterate directly over each item of a given **iterable** object (such as a sequence)
What if We Don’t Know When to Stop?

• We always know the stopping condition of a `for` loop: when there are no more elements in the sequence

  `["A", "chilly", "autumn", "day"]`

• Are there contexts where we don’t know when to stop a loop?
  • Suppose you want to play a "guessing game" where a user repeatedly guesses numbers until they correctly guess the secret number

  • How many times should the loop execute?
  • Under what condition should the loop end?
The While Loop

A `while` loop executes the loop body 0 or more times, stopping once the loop condition evaluates to `False`:

```python
while <boolean expression>:
    <loop body>
    <loop body>
...
```

- `while False:`
  - print("never enters")

- `while True:`
  - print("never leaves")

Loop body never executes

"Infinite" loop!
While Loop Example

- Example of a `while` loop that depends on user input:

```python
prompt = "Please enter a name (type quit to exit): "
name = input(prompt)

while (name != "quit"):  
    print("Hi,", name)  
    name = input(prompt) 

print("Goodbye")
```
While Loop Example: Print Halves

- Given a number, print all the positive “halves”: keep dividing \( n \) by 2 and printing the quotient until it becomes smaller than 0

```python
def print_halves(n):
    while n > 0:
        print(n)
        n = n//2

print_halves(100)
```
While Loop to Print Halves

• Given a number, print all the positive “halves”: keep dividing \( n \) by 2 and printing the quotient until it becomes smaller than 0.

```python
def print_halves(n):
    while n > 0:
        print(n)
        n = n//2

print_halves(100)
```

```python
def print_halves2(n):
    while n > 0:
        print(n)
        n = n/2

print_halves2(100)
```
While Loop to Print Halves

• Given a number, print all the positive “halves”: keep dividing $n$ by 2 and printing the quotient until it becomes smaller than 0

```python
def print_halves(n):
    while n > 0:
        print(n)
        n = n//2

print_halves(100)

def print_halves2(n):
    while n > 0:
        print(n)
        n = n/2

print_halves2(100)
```

Float division! Be careful!
While Loop to Print Halves

• Given a number, print all the positive “halves”: keep dividing $n$ by 2 and printing the quotient until it becomes smaller than 0

```python
def print_halves(n):
    while n > 0:
        print(n)
        n = n//2

print_halves(100)
```

```python
def print_halves3(n):
    while n > 0:
        print(n)
        n = n//2

print_halves3(100)
```
While Loop to Print Halves

- Given a number, print all the positive “halves”: keep dividing $n$ by 2 and printing the quotient until it becomes smaller than 0

```python
def print_halves(n):
    while n > 0:
        print(n)
        n = n//2

print_halves(100)
```

```python
def print_halves3(n):
    while n > 0:
        print(n)
        n = n//2

print_halves3(100)
```

Another infinite loop! Indentation matters!
**while** and **if** side by side

**if** boolean_expression:

# statement 1
# statement 2
....
....

# end of if

**while** boolean_expression:

# statement 1
# statement 2
....
....

# end of while

 Execute body **once** if the **boolean expression** evaluates to true

 Keep executing body **as long as** the **boolean expression** (**continues**) to evaluate to true
Side by Side: for and while loops

Iteration steps are **implicit** in a Python for loop: \( i \) takes on values 0, 1, 2, 3, 4

```python
for i in range(5):
    print('$' * i)
```

Explicitly **initialize** variable

```python
i = 0
while i < 5:
    print('$' * i)
    i += 1
```

**Test** stopping condition

**Common** while loops steps that we **explicitly** write:

- **Initialize** a variable used in the test condition
- **Test** condition that causes the loop to end when **False**
- Within the loop body, **update** the variable used in the test condition
Breaking out of loops

• Stopping condition of for loop: **no more elements in sequence**

• What if we want to stop our iteration early: how did we handle this?
  • return (or, less ideally, break)

• Let's examine an example: **index_of(elem, l)**
  • Write a function **index_of(elem, l)** that takes two arguments (**elem** of any type and list **l**) and returns the first index of **elem** if **elem** is in the list **l** and **-1** otherwise

```python
>>> index_of('blue', ['red', 'blue', 'blue'])
1
>>> index_of(14, [23, 1, 10, 11, 14])
4
>>> index_of('a', ['b', 'c', 'd', 'e'])
-1
```
def index_of(elem, l):
    for i in range(len(l)):
        # match?
        if l[i] == elem:
            # stop loop!
            return i
    # if not found
    return -1

def index_of(elem, l):
    found = False  # flag
    index_of_elem = -1
    i = 0
    while not found and i < len(l):
        # match?
        if elem == l[i]:
            # stop the loop!
            found = True
            index_of_elem = i
        # keep going
        i += 1
    return index_of_elem
Mutability
Lists are Mutable

• Lists are a mutable data type in Python:
  • After a list is created, we can change its value
• There are many ways to mutate a list, we will only discuss two of these for now (we'll examine others after the midterm)
  • Direct assignment (e.g., lst[index] = item)
  • Appending to list using .append(item) notation
Direct Assignment

• Lists are a **mutable** data type in Python:
  • After a list is created, we can **change** its value
  • One way to modify a list is by **direct assignment**

```python
>>> my_list = ['cat', 'dog']
>>> my_list[1] = 'fish'
>>> my_list
['cat', 'fish']
```

*my_list has changed!*
Direct Assignment

An assignment operation to an **existing** index of a list changes the value stored at that index.

Syntax: `my_list[index] = item`

```python
>>> my_list = ['cat', 'dog']
>>> my_list[1] = 'fish'
>>> my_list
['cat', 'fish']
>>> my_list[7] = 'oops'
IndexError: list assignment index out of range
```
Using `.append(item)`

Appending to a list places a new item *after* the current end of the list, increasing the list's length by one.

**Syntax:**  
```python
my_list.append(item)
```

**Example.**

```python
my_list = [1, 7, 3, 4]
my_list.append(5)  # insert 5 after the end of list
```

```
myList Before
[1, 7, 3, 4]
myList After
[1, 7, 3, 4, 5]
```
Sneaky Appending

• We've often updated "accumulator lists" by "appending" items in loops

• So far we have been using `+=` (concatenation)

  • `var += val` normally is a shorthand for `var = var + val`

  • But when `var` is a list, Python **secretly** calls `var.append(val)`

```
>>> my_list = ['cat', 'dog']
>>> my_list += ['fish']
>>> my_list
['cat', 'dog', 'fish']
```

Python actually replaces `+=` with `append` without telling us!
Explicit Appending

- If we instead explicitly use the `.append(item)` syntax, then the code we execute is the code that we actually wrote.
- This also avoids one of the recurring errors that we've been running into in our labs! (Type mismatches with `+=`)

```python
>>> my_list = ['cat', 'dog']
>>> my_list += ['fish']
>>> my_list
['cat', 'dog', 'fish']

>>> my_list = ['cat', 'dog']
>>> my_list.append('fish')
>>> my_list
['cat', 'dog', 'fish']
```

Brackets needed here

NO brackets here
Other data types we have seen are immutable

- Strings, ints, floats, range() are immutable
- Once created, we cannot change the value of an immutable data type

```python
>>> my_string = 'cat'
>>> my_string[0] = 'b'
```

Will this let us change `my_string` to 'bat'?

```
TypeError                                 Traceback (most recent call last)
Cell In[25], line 2
  1 my_string = 'cat'
----> 2 my_string[0] = 'b'

TypeError: 'str' object does not support item assignment
```

Cannot change a string!
Mutability has Consequences!

- Mutability of data types can have **unintended consequences**

```python
>>> word = "hello"
>>> copy = word
>>> word = word + "world"
>>> copy
"hello"
```

```python
>>> word_list = ["hello"]
>>> copy = word_list
>>> word_list.append("world")
>>> copy
['hello', 'world']
```

Changing `word` does not change `copy`

Changing `word_list` also **changes** `copy`
Mutability has Consequences!

- Mutability of data types can have unintended consequences
- **Aliasing as a consequence of Mutability.** In Python, creating a `copy` of a mutable object creates an alias rather than a true copy

```python
>>> word = "hello"
>>> copy = word
>>> word = word + "world"
>>> copy
"hello"

>>> word_list = ["hello"]
>>> copy = word_list
>>> word_list.append("world")
>>> copy
["hello", "world"]
```

Changing `word` does not change `copy`

Changing `word_list` also changes `copy`
Takeaways

• New iteration statement: **while** loop as an alternative to **for** loops are meant to iterate for a fixed number of times
  
  • Used when the stopping condition is determined "on the fly"
  
  • Keeps iterating as long as Boolean condition evaluates to **True**

• Lists are mutable data types
  
  • Can modify the contents of a list by direct assignment or by using .append()

• Strings, ints, floats, range() are immutable: cannot be modified

• Mutability has consequences!
  
  • Will discuss **aliasing** in detail next lecture
Modules vs Scripts
Importing Functions vs Running as a Script

• **Question.** If you only have function definitions in a file `funcs.py`, and run it as a script, what happens?

  ```
  % python3 funcs.py
  ```

• For testing functions, we want to call /invoke them on various test cases, in Labs, we do this in a separate file called `runtests.py`

  • To add function calls in `runtests.py`, we put them inside the guarded block `if __name__ == "__main__":`

  • The statements within this special guarded are only run when the file is run as a *script* but not when it is imported as a *module*

• Let's see an example
# foo.py
# test the role of __name__ variable
print("__name__ is set to", __name__)

Running foo.py as a script

shikhasingh@Shikhas-iMac cs134 % python3 foo.py
__name__ is set to __main__

shikhasingh@Shikhas-iMac cs134 % python3
Python 3.10.0 (v3.10.0:b494f5935c, Oct 4 2021, 14:59:20) [Clang 12.0.5 (clang-1205.0.22.11)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import foo
__name__ is set to foo

Importing it as a module
Takeaway: `if __name__ == "__main__"`

- If you want some statements (like test calls) to be run ONLY when the file is run as a script
  - Put them inside the guarded `if __name__ == "__main__"` block

- When we run our automatic tests on your functions we import them and this means name is NOT set to main
  - So nothing inside the guarded `if __name__ == "__main__"` block is executed

- This way your testing /debugging statements do not get in the way