Admin

• Lab 2 marks posted
  – If you think there’s an error in recorded mark, talk with TAs in lab first to resolve

• Lab 5 posted
  – Last lab before midterm
  – Practice with conditionals, loops, writing some code from scratch

• Jessica office hours rescheduled today
  – 3-4pm instead of usual 11:30am – 12:30pm
Vocabulary: Python "Blocks"

- In Python, a program is constructed from a collection of statements (or commands)
  - A statement may contain one or more expressions
  - In Scratch, both statements and expressions are called blocks
- In Python, a collection of one or more statements with the same level of indentation is called a block
  - Any Python statement that ends in a colon “:” must be followed by an indented block
  - In the `def` statement, that block becomes the body of the function and will be executed when the function is called
  - In the `if`/`elif`/`else` conditional statements, the blocks are executed only when the corresponding conditions are True
  - In the `for`/`while` looping statements, the blocks are executed zero or more times
Assertions and Early Failure

• An assertion is a way of stopping Python

\[
\text{assert}(\text{income} \geq 0)
\]

  – If the condition is False, Python will immediately stop executing with an AssertionError

• Used to ensure that a certain property holds
  – Also signals to other programmers that you believe this property should always hold
  – For example: Ensure that function arguments satisfy preconditions

• Considered a rather brutal way of dealing with the unexpected
  – Use it only when you expect the condition to be very, very unlikely
  – An alternative: If a function's arguments do not satisfy the precondition you could return `None`
  – There are other ways of signalling unexpected situations: Exceptions

• However, better to use an assertion than to ignore a bad situation
Repetition: for Loop

- Like Scratch's repeat block, but more general
- Form of a for loop:

```python
for variable in sequence :
    block
```

where

- `variable` is a new variable name just declared there, and
- `sequence` is a sequence (a list) of values, like a string, or a sequence created by a call to `range(v1, v2)`
- the block will be executed once for each value in the `sequence` and each time `variable` will be assigned that value

- Example:

```python
for char in "Hello":
    print(char)
```

will print: `Hello`
The range() Function

- Often used in for-loops. Has various forms
- `range(start, stop)` creates a sequence of integers from start to stop-1
  - `range(3, 9)` returns [3, 4, 5, 6, 7, 8]
- `range(stop)` creates a sequence of integers from 0 to stop-1
  - `range(5)` returns [0, 1, 2, 3, 4]
  - `range(0)` returns [] (the empty sequence)
- `range(start, stop, step)` creates a sequence of integers from start to stop-1, where each integer is produced by the previous one by adding the step
  - `range(2, 10, 3)` returns [2, 5, 8]
  - `range(2, 11, 3)` returns [2, 5, 8]
  - `range(5, -5, -2)` returns [5, 3, 1, -1, -3]
Testing the results of the \texttt{range()} function

- Note that \texttt{range} does not evaluate in the console the way that you might expect
  - Entering:
    \begin{verbatim}
    >>> range(0,3)
    \end{verbatim}
  - Produces\texttt{range(0, 3)}

- If you want to test out the \texttt{range} function in the console to see what sequence it produces, you have to use \texttt{list()}
  \begin{verbatim}
  >>> list(range(0,3))
  \end{verbatim}
  \texttt{[0, 1, 2]}

- We will talk in detail about lists after the midterm
  - Except for the above purpose, don’t worry about it for now.
for Loop Examples

• Example 2: Consider the following function:

```python
def sum_upto( n):
    '''(int) -> int
    ...'''
    sum = 0
    for num in range(1, n+1):
        sum += num
    return sum
```

– What does this function compute?
– What is the value of `sum_upto(6)`?

• Example 3: What does this function do?

```python
def sum2_upto( n):
    '''(int) -> int
    ...'''
    sum = 0
    for num in range(1, n+1):
        sum += num * num
    return sum
```
while Loops

• Like Scratch's ‘repeat until’ block
• Allows you to repeat a block of statements as long as a specified condition is **True**
• Form of the while-loop

```
while condition:
  block
```

– where **condition** is an expression that returns a boolean value (**True** or **False**)
– if condition is False the block is skipped and the loop terminates
– if condition is True the block is executed;
  • then the condition is tested again and if it is True, the block is executed again, and so on until the condition becomes False and the loop terminates
– if the condition never becomes false the loop goes on forever (like Scratch's forever block) and it is called an **infinite loop**
while Loop Example 1

• Example 1:
  ```python
  times = int(input("Enter times to cheer: "))
  while times != 0:
      print("hip, hip, hurray!")
      times -= 1
  print("That's it!")
  ```

• If the user input is 3, it will print:
  ```
  hip, hip, hurray!
  hip, hip, hurray!
  hip, hip, hurray!
  That's it!
  ```

• What the program will do if the input is -1?
  – How can we fix it?
What does the following function do?

```python
def population(init_population, birth_rate, death_rate, periods):
    ''(int, int, int, int) -> int
    It calculates ......
    init_population : the number of organisms when the experiment starts
    birth_rate, death_rate : the birth and death rates from one period to next
        given in percentages i.e. 10 means 10%
    periods : the number of time periods we want to run the experiment for
    :
    ::
    cur_population = init_population
    cur_period = 0

    while cur_period < periods :
        # Estimate the population of next period
        cur_period = cur_period + 1
        cur_growth = cur_population *(birth_rate -
            death_rate)/100
        cur_population = cur_population + round(cur_growth)
    return cur_population
```

What does the call `population(1000, 20, 10, 1)` return?
Controlling Loops

- Sometimes we need to break a loop before the loop terminates on its own.
- In this case we use a `break` statement which makes the program to exit the loop.
- Example: Suppose we want to compute the average of all the numbers the user gives us until a 0 is typed:

```python
count = 0
sum = 0
while True:
    n = float(input("Enter next number or 0 to stop: "))
    if n == 0:
        break
    sum = sum + n
    count = count + 1
print ("The average of the", count, "numbers is", sum/count)
```
Sometimes we need to skip the current iteration and continue with the next iteration in a loop (i.e. jump to the top of the loop).

In this case we use a `continue` statement.

Example: As before, but we want to skip any negative number:

```python
count = 0
sum = 0
while True :
    n = float(input("Enter next number or 0 to stop: "))
    if n == 0 :
        break
    if n < 0 :
        continue
    sum = sum + n
    count = count + 1
print("Average of positives is", sum/count)
```
Avoiding break and continue

• It is often convenient to use break and/or continue, but if you can rearrange the code and avoid them, the code may be more readable.

• A loop to compute the average of all positive numbers entered before a zero (without break and continue):

```python
count = 0
sum = 0
n = True  # Any value other than 0 or 0.0 works.
while n != 0:
    n = float(input('Enter number: '))
    if(n > 0):
        sum += n
        count += 1
print('Average of positives is: ', sum / count)
```
Conclusion

• Python provides all the common relational (comparison) operators (==, ! =, <=, <, >=, >, in) and boolean operators (and, or, not).

• Values 0.0, 0, '', None, False are all false. True and any other value is true.

• To control the flow of statement execution, we can use: Choice statements:
  
  if
  if-else
  if–elif–else

  and repetition statements or loops:
  for-loop
  while – loop

• Using these we can write programs to solve very complex problems.