CPSC 301: Computing in the Life Sciences
Lecture Notes 6:
Control Flow Statements

pre-class notes

Jessica Dawson
jqdawson@cs.ubc.ca
http://www.cs.ubc.ca/~jqdawson

University of British Columbia
Department of Computer Science

2016 W2
Objectives

When you complete this section you should be able to:

• Read, trace, modify and develop programs that use relational operators, Boolean values and Boolean operators.

• Read, trace, modify and develop programs that make choices using \texttt{if} / \texttt{elif} / \texttt{else} conditional statements.

• Use the \texttt{assert} statement to infer or enforce properties about data (and fail early if those properties are not true).

• Read, trace, modify and develop programs that use simple types of repetition: \texttt{for} and \texttt{while} loops.

• Read, trace, modify and develop programs that use \texttt{break} and \texttt{continue} statements.

• Read, trace, modify and develop programs that use combinations of all of the above, including nested loops and conditionals.
Boolean Values and Relational Operators

- Python uses the key words `True` and `False` to represent the two Boolean values.
- Can compare numbers and strings with the following relational operators (like in Scratch). Each comparison returns True or False:

  - `>`, `<`, `>=`, `<=`, `==` (equal), `!=` (not equal)
  - `5 < 3` is False,
  - `5.5 <= 20` is True,
  - `'hello' <= 'world'` is True
  - `'hello' != 'world'` is True,
  - `5 == (2 + 3)` is True
Boolean Operators

• **Boolean operators**: `and`, `or`, and `not`
  – `a and b` : True if `a` and `b` are both True; False otherwise
  – `a or b` : False if `a` and `b` are both False; True otherwise
  – `not a` : True if `a` is False and False if `a` is True

• Can combine Boolean operators with relational operators (like Scratch) Example:
  • if `x` is 5 and `y` is 10 then `x >= y or y == 10` is True
  • if `x` is 1 and `y` is -1 then `x >= y and y == 10` is False
Number and Strings as Boolean Values

• In Python we can apply boolean operators to numbers and strings
• Python converts other values to Booleans as following:
  – 0, 0.0, the empty string, and None are treated as False
  – any other value is treated as True
• Examples:
  – 0 and True is 0
  – 0 and 3 is 0
  – 0 or 3 is 3
• Nevertheless, in order to make programs easy to understand we should avoid mixing different types of values that don't make sense.
  – We generally recommend you avoid using statements like the above.
More on Boolean Values

• We can check if a string is a substring of another string using the `in` operator:
  - "gc" in "aaggagacgccccat" is True
  - "ggcc" in "aaggagacgccccat" is False

• We can assign Boolean values to variables
  - `speed = 100`
  - `high = speed > 100` (then high is False)

• We can define functions that return Boolean values:
  ```python
def between(value, low, high):
    """ returns True if value is strictly between low and high"
    return low < value and value < high
  ```

  ```python
val = 15
between(val, -20, 20) returns True
between(val, -20, 15) returns False
```
Boolean Operation Gotchas

• Test for equality \(==\) is different from assignment \(=\)
  – The former is a \textit{relational operator} forming part of an \textit{expression} that evaluates to \texttt{True} or \texttt{False}
  – The latter is an \textit{assignment operator} forming part of a \textit{statement} which assigns a value to a variable; statements have no value of their own

• Be careful about \textit{operation precedence}
  – If in doubt, just use parentheses!

• Recommend you avoid combining comparisons
  – eg: use \(0 < x \text{ and } x <= 42\), avoid \(0 < x <= 42\)

• Be careful about comparing strings
  – eg: \('XYZ' < 'abc', 'a*c' < 'abc', 1 < 'a'\)
  – String comparison is obvious as long as the strings are all of a single case (upper or lower) and contain no numbers or symbols
Control Flow Statements

• Python has similar control structures (and several more) like those we have seen in Scratch
  – Choice statements like `if/elif/else`
  – Loops like `for–loop, while–loop`

• Each of these statements always ends with a `:` (colon) followed by a block of statements (also called compound block)

• Statements within the compound block are indented
  – All statements in the same block use the same indentation (typically 4 spaces) unless there are sub blocks
  – Good style: use the same amount of indentation throughout a file, and indent comments to match the statements
  – Compound block ends when indentation ends

• Indentation provides a visual cue similar to Scratch’s clamp shaped control blocks
  – In Python, correct indentation is mandatory
  – In most other programming languages, compound statements are explicitly delimited and indentation is a style choice
Making Choices

• Form of an if-statement:
  ```python
  if condition:
    block
  or
  if condition:
    if-block
  else:
    else-block
  ```
  where each block is a sequence of statements which are indented

• condition can be a boolean or any other value. In the second case:
  – a value 0, 0.0, None and "" (the empty string) are treaded as False
  – any other value is treaded as True

• Example:
  ```python
  if total_purchase >= 1000:
    print("You are eligible for 20% discount")
  else:
    print("You are eligible for 10% discount")
  ```
Nested if-Statements

Use **elif** to combine **else** and **if** statements

Instead of:

```python
def standing(grade):
    if grade >= 80:
        return "1st class"
    else:
        if grade >= 65:
            return "2nd class"
        else:
            if grade >= 50:
                return "pass"
            else:
                return "fail"
```

We write:

```python
def standing(grade):
    if grade >= 80:
        return "1st class"
    elif grade >= 65:
        return "2nd class"
    elif grade >= 50:
        return "pass"
    else:
        return "fail"
```
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

  ```python
  assert(income >= 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:

\[
\text{for variable in sequence :}
\]

\[
\text{block}
\]

where

– \text{variable} is a new variable name just declared there, and

– \text{sequence} is a sequence (a list) of values, like a string, or a sequence created by a call to \text{range(v1, v2)}

– the block will be executed once for each value in the \text{sequence} and each time \text{variable} will be assigned that value

• Example:

\[
\text{for char in "Hello": }
\]

\[
\text{print(char)}
\]

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
    \begin{verbatim}
    range(0, 3)
    \end{verbatim}

- 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}
  \begin{verbatim}
  [0, 1, 2]
  \end{verbatim}

- 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

```python
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?
Example 2

What does the following function do?

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:

```
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)
```
Controlling Loops (cont'd)

- 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 :  # added break to stop when 0 is entered
        break
    if n < 0 :    # added continue to skip negative numbers
        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.