COMP 364: Computer Tools for Life Sciences
Python programming: Control flow: for loops, while loops

Christopher J.F. Cameron and Carlos G. Oliver
Quiz #1

Score Distribution:

Percentage of Users

Class Average: 83.81% (Std Dev = 12.29%)
Key course information

Assignment #1 is now available!
- Available through the course website: http://cs.mcgill.ca/~cgonza11/COMP_364/
- Due: September 29th at 11:59:59 pm
- Start early, do better!

Assignment topics
- Do you have an idea for a COMP 364 assignment? Or would you like a specific area of bioinformatics covered?
- Let us know on MyCourses
  - Discussions → General Discussion → Assignment Topic Suggestions
What is an iterable?

To iterate over a sequence means to visit each element of the sequence, and do some operation for each element.

In Python, we say that an object is an iterable when your program can iterate over it.

- Or an iterable is an object that represents a sequence of one or more values.

All instances of Python’s sequence types are iterables:

- Lists
- Strings
- Tuples
All **iterables** can be passed to the **`iter()`** to get an **iterator**

```python
1   iter(['some', 'list'])
2   # output: <list_iterator object at 0x7f227ad51128>
3   iter('some string')
4   # output: <str_iterator object at 0x7f227ad51240>
```

Okay...but what's an **iterator**?

- Iterators perform a singular function
  - To return the next item in an iterator
Iterators can be passed to `next()` to get their next item

```python
iterator = iter("hi")
next(iterator)
# output: 'h'
next(iterator)
# output: 'i'
next(iterator)
# StopIteration exception is thrown
```

If there is no next item, Python will raise a `StopIteration` exception
Iterators are also iterables

Passing an **iterable** to `iter()` gives us an **iterator**

Calling `next()` on an **iterator** gives us the next item or raises a `StopIteration` exception

**Iterators** can be passed to `iter()` to get the **iterator** back

```python
iterator = iter("hi")
iterator_2 = iter(iterator)
iterator is iterator_2
# output: 'True'
```

This means **iterators** are also **iterables**
Python membership operators

**in**

- Evaluates to true if it finds a variable in the specified sequence and false otherwise

```python
iterator = iter(['BRCA1', 'MYH7', 'ABO'])

"BRCA1" in iterator # output: 'True'
"Myc" in iterator    # output: 'False'
```

**not in**

- Evaluates to true if it does not find a variable in the specified sequence and false otherwise

```python
"BRCA1" not in iterator # output: 'False'
"Myc" not in iterator   # output: 'True'
```
The `for` statement in Python calls `iter()` and `next()` automatically.

```python
sequence = ["BRCA1","MYH7","ABO"]

for gene_name in sequence:
    print(gene_name)

#output:
# BRCA1
# MYH7
# ABO
```

`for` statements allows us to implement iteration more easily.

- How do they work?
sequence = ["BRCA1","MYH7","ABO"]
for gene_name in sequence:  # iter(sequence)
    print(gene_name)
    #output:
    #   BRCA1
    #   MYH7
    #   ABO

'gene_name' is called the **loop variable**
- Value changes to the next item in sequence for each iteration of the `for` loop
For loops #3

```python
sequence = ["BRCA1", "MYH7", "ABO"]
for gene_name in sequence:  # iter(sequence)
    print(gene_name)

#output:
#  BRCA1
#  MYH7
#  ABO
```

**in**, in this case, is not a Python membership operator.

- At the start of a loop, the Python interpreter evaluates the object after **in** and expects an iterator.
sequence = ["BRCA1","MYH7","ABO"]

for gene_name in sequence:  # iter(sequence)
    print(gene_name)

#output:
#   BRCA1
#   MYH7
#   ABO

Line 3 is the **loop body**

- Always indented relative to the *for* statement
- The **loop body** is performed for each item in the sequence
For loops #5

```python
sequence = ["BRCA1","MYH7","ABO"]
for gene_name in sequence:  # iter(sequence)
    print(gene_name)
#output:
#   BRCA1
#   MYH7
#   ABO
```

On each iteration or pass of the loop:

- A conditional check (**terminating condition**) is done to see if there are more items to be processed
- If there are none left, **StopIteration** exception is thrown
- Execution continues to next statement after the loop body
While loops

The **while** loop statement in Python repeatedly executes its target statement(s) as long as the given condition is true.

```python
1  while condition:
2    statement(s)
```

Statement(s) can be a single statement or block of statements.

The **condition** may be an *expression* and **True** is any non-zero value.

The loop iterates while the condition is **True**.

When the condition becomes false, the loop ends and execution continues to the next statement after the loop.
While loops #2

One way to implement our previous `for` loop as a `while` loop

```python
sequence = ["BRCA1","MYH7","ABO"]
while len(sequence) > 0:
    print(sequence.pop())
#output:
#  ABO
#  MYH7
#  BRCA1
```

The order is reversed

- Why?
- How can we fix this?
while loop that maintains previous gene_name order

```python
index = 0
sequence = ["BRCA1","MYH7","ABO"]
N = len(sequence)-1
while index <= N:
    print(sequence[index])
    index += 1

#output:
# BRCA1
# MYH7
# ABO
```
Infinite loops

A loop becomes an infinite loop if a condition never becomes **False**

```python
# example 1
while True:
    statement(s)

# example 2
var = 1
count = 0
while var == 1:
    count += 1
print(count)
```
Else clause on loop statements

A confusing aspect of Python

```python
if True:
    print("Then")
else:
    print("Else")
# Output:
# 'Then'

for item in [1]:
    print("Then")
else:
    print("Else")
# Output:
# 'Then'
# 'Else'
```
```python
sequence = ['BRCA1']
while sequence:
    print('Then')
    sequence.pop()
else:
    print('Else')
# Output:
# 'Then' (then 'BRCA1' from pop.())
# 'Else'
sequence = ['BRCA1']
if sequence:
    print('Then')
else:
    print('Else')
# Output:
# 'Else'
```
range() is a built-in Python function that returns an iterator

- In Python 3.x, `range()` replaces Python 2.x’s `xrange()`

`range([start], stop[, step])`

- **start**: starting number of the sequence
- **stop**: generate numbers up to, but not including this number
- **step**: difference between each number in the sequence

All parameters to `range()`:

- Must be integers
- Can be positive or negative
- 0-index based
Using range()

```python
for index in range(2):
    print(index)

#output:
# 0
# 1

for index in range(1,3):
    print(index)

#output:
# 1
# 2

for index in range(6,10,3):
    print(index)

#output:
# 6
# 9
```
enumerate()

enumerate(iterable, start=0)

- A built-in Python function that returns an **enumerate object**
- Enumerate object is iterable
- Calling `next()` returns a tuple containing a count and value
- Count begins at `start` (default=0)

```python
sequence = ['BRCA1', 'MYH7', 'ABO']
list(enumerate(sequence))
# outputs: [(0, 'BRCA1'), (1, 'MYH7'), (2, 'ABO')]
```

What happens if you don’t pass the enumerate object to `list()`?
Using enumerate()

```python
sequence = ["BRCA1","MYH7","ABO"]
for index, item in enumerate(sequence):
    print(index, item)
# outputs:
# 0 BRCA1
# 1 MYH7
# 2 ABO

for index, item in enumerate(sequence, 2):
    print(index, item)
# outputs:
# 2 BRCA1
# 3 MYH7
# 4 ABO
```