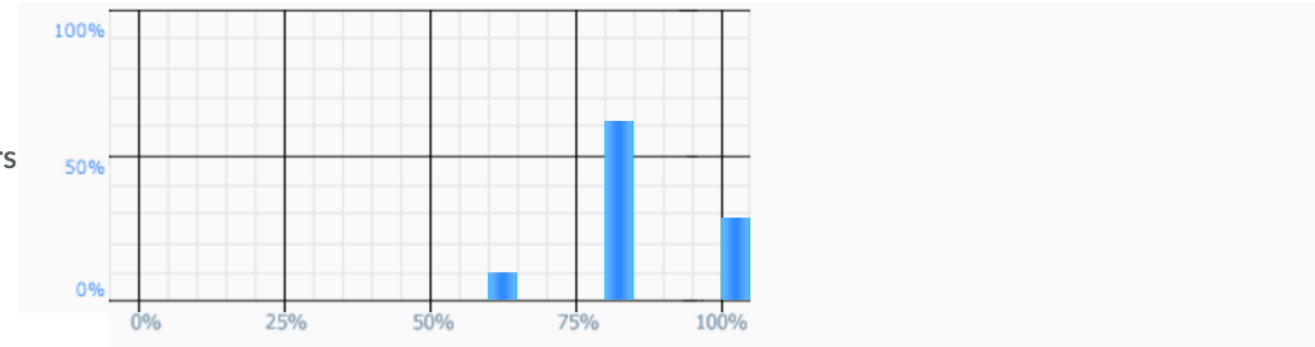
COMP 364: Computer Tools for Life Sciences

Python programming: Control flow: for loops, while loops

Christopher J.F. Cameron and Carlos G. Oliver

Quiz #1

Score Distribution:



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

Python iterators

All **iterables** can be passed to the **iter()** to get an **iterator**

```
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

Python iterators #2

Iterators can be passed to **next()** to get their next item

```
1       iterator = iter("hi")
2       next(iterator)
3       # output: 'h'
4       next(iterator)
5       # output: 'i'
6       next(iterator)
7       # 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

```
1    iterator = iter("hi")
2    iterator_2 = iter(iterator)
3    iterator is iterator_2
4    # 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

```
iterator = iter(["BRCA1","MYH7","ABO"])
"BRCA1" in iterator # output: 'True'
"Myc" in iterator   # output: 'False'
```

not in

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

```
"BRCA1" not in iterator # output: 'False'
"Myc" not in iterator   # output: 'True'
```

For loops

The **for** statement in Python calls **iter()** and **next()** automatically

```
1       sequence = ["BRCA1","MYH7","ABO"]
2       for gene_name in sequence: # iter(sequence)
3           print(gene_name)
4       #output:
5       #    BRCA1
6       #    MYH7
7       #    ABO
```

for statements allows us to implement iteration more easily

- How do they work?

For loops #2

```
1       sequence = ["BRCA1","MYH7","ABO"]
2       for gene_name in sequence: # iter(sequence)
3           print(gene_name)
4       #output:
5       #    BRCA1
6       #    MYH7
7       #    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

```
1       sequence = ["BRCA1","MYH7","ABO"]
2       for gene_name in sequence: # iter(sequence)
3           print(gene_name)
4       #output:
5       #    BRCA1
6       #    MYH7
7       #    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

For loops #4

```
1       sequence = ["BRCA1","MYH7","ABO"]
2       for gene_name in sequence: # iter(sequence)
3           print(gene_name)
4       #output:
5       #    BRCA1
6       #    MYH7
7       #    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

```
1       sequence = ["BRCA1","MYH7","ABO"]
2       for gene_name in sequence: # iter(sequence)
3           print(gene_name)
4       #output:
5       #    BRCA1
6       #    MYH7
7       #    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 it's target **statement(s)** as long as the given condition is true

```
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

```
1       sequence = ["BRCA1","MYH7","ABO"]
2       while len(sequence) > 0:
3           print(sequence.pop())
4       #output:
5       #    ABO
6       #    MYH7
7       #    BRCA1
```

The order is reversed

- Why?
- How can we fix this?

While loops #3

while loop that maintains previous gene_name order

```
1       index = 0
2       sequence = ["BRCA1","MYH7","ABO"]
3       N = len(sequence)-1
4       while index <= N:
5           print(sequence[index])
6           index += 1
7       #output:
8       #    BRCA1
9       #    MYH7
10      #    ABO
```

Infinite loops

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

```
1       # example 1
2       while True:
3           statement(s)
4       # example 2
5       var = 1
6       count = 0
7       while var == 1;
8           count += 1
9           print(count)
```

Else clause on loop statements

A confusing aspect of Python

```
1     if True:
2         print("Then")
3     else:
4         print("Else")
5     # Output:
6     # 'Then'
7
8     for item in [1]:
9         print("Then")
10    else:
11        print("Else")
12    # Output:
13    # 'Then'
14    # 'Else'
```

```
    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")
    # Ouput:
    # 'Else'
```

range()

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()

```
1     for index in range(2):
2         print(index)
3     #output:
4     #   0
5     #   1
6     for index in range(1,3):
7         print(index)
8     #output:
9     #   1
10    #   2
11    for index in range(6,10,3):
12        print(index)
13    #output:
14    #   6
15    #   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)

```
1    sequence = ["BRCA1","MYH7","ABO"]
2    list(enumerate(sequence))
3    # ouptuts: [(0, 'BRCA1'), (1, 'MYH7'), (2,'ABO')]
```

What happens if you don't pass the enumerate object to **list()**?

Using enumerate()

```
1   sequence = ["BRCA1","MYH7","ABO"]
2   for index, item in enumerate(sequence):
3       print(index, item)
4   # ouputs:
5   # 0 BRCA1
6   # 1 MYH7
7   # 2 ABO
8
9   for index, item in enumerate(sequence,2):
10      print(index, item)
11  # ouputs:
12  # 2 BRCA1
13  # 3 MYH7
14  # 4 ABO
```