BINF2111 – Introduction to Bioinformatics Computing

If else or not if else – the conditional paradox

Richard Allen White III, PhD
RAW Lab

Lecture 20 – Thrusday Nov 11th, 2021

Learning Objectives

- docstrings

- Conditionals (If/then)

- Conditionals (Else if)

- Conditionals (Or else)

- Conditionals (Continue)

- Quiz 20

Docstrings

Docstrings allow you to print out the first string right after the defined function.

```
#Function definition is here
def printme( str ):
	“This prints a passed string into this function”
	print str;
	return;

#Calling the printme function
printme(“First call of user-defined function!")
printme(“Second call of user-defined function")

print(printme.__doc__)
```

Docstrings

Docstrings allow you to print out the first string right after the defined function.

```
#Function definition is here
def printme( str ):
    “”” This prints a passed string into this function
        I wrote this
    ”””
    print str;
    return;

#Calling the printme function
printme(“First call of user-defined function!")
printme(“Second call of user-defined function")

print(printme.__doc__)
```

Conditionals

Conditions – Conditions are code that produce a true or false answer.

assert printme(“this string”) == “this string”

This is a “true” statement.

assert printme(“this string”) == “wrong string”

This is a “false” statement.

Conditionals

Conditions – Conditions are code that produce a true or false answer.

We’ve seen some of the conditions before when using while loops.

equals (represented by **==**)

greater and less than (represented by **>** and **<**)

greater and less than or equal to (represented by **>=** and **<=**)

not equal (represented by **!=**)

is a value in a list (represented by **in**)

are two objects the same (represented by **is**)

Conditionals

1. print(3 == 5)
2. print(3 > 5)
3. print(3 <=5)
4. print(len("ATGC") > 5)
5. print("GAATTC".count("T") > 1)
6. print("ATGCTT".startswith("ATG"))
7. print("ATGCTT".endswith("TTT"))
8. print("ATGCTT".isupper())
9. print("ATGCTT".islower())
10. print("V" in ["V", "W", "L"])

Conditionals

1. `print(3 == 5)`
2. `print(3 > 5)`
3. `print(3 <=5)`
4. `print(len("ATGC") > 5)`
5. `print("GAATTC".count("T") > 1)`
6. `print("ATGCTT".startswith("ATG"))`
7. `print("ATGCTT".endswith("TTT"))`
8. `print("ATGCTT".isupper())`
9. `print("ATGCTT".islower())`
10. `print("V" in ["V", "W", "L"])`

Although we are using the print function, it is not printing the actual text, it is printing the special value that we are testing. True or False.

Conditionals

Print(True)

Print(False)

These print without quotations, because they are special values.

Conditionals

```
Print(True)
Print(False)
```

These print without quotations, because they are special values.

Conditionals

Programs have to make decisions

```
for line in file.readlines():
	if line.startswith(">"):
```

Line 1 initiates a loop using lines in a file

Line 2 is a conditional test

Conditionals

Programs have to make decisions

```
for line in file.readlines():
        if line.startswith(">"):
```

Line 1 initiates a loop using lines in a file

Line 2 is a conditional test

Operators

Commonly used operators

```
+, -, *, /, **, %
+=, -=, /=, **=                              #there is no ++ operator
equality ==
greater than, less than >, <
greater than/equal to etc >=, <=
not-equality !=
```

```
    temp = 5
    temp += 2          #can be written as: temp = temp+2
        print(temp)    #the += does operation and assigns 7
```

Operators

Commonly used operators

equality ==

greater than, less than >, <

greater than/equal to etc >=, <=

not-equality !=

in/not in (example: if a in mylist:)

is/not is -- values on either side of the operator point to the same value

Operators

Commonly used operators

equality ==

greater than, less than >, <

greater than/equal to etc >=, <=

not-equality !=

in/not in (example: if a in mylist:)

is/not is -- values on either side of the operator point to the same value

Logical Operators

Commonly used operators

AND – both must evaluate True

OR – one or the other must evaluate True

NOT – must evaluate False

True and False are not strings. They are special values reserved in Python.

string methods like .startswith(), .islower() -- they are either true or not true, they evaluate something in the string rather than counting or manipulating

Conditionals: If/then

If this	Else if that	Or else
Do this	Do that	Do nothing
And then this	And that other thing	Exit

Conditionals: If/elif/else

```
if condition_1:
    statement_block_1
elif condition_2:
    statement_block_2
else:
    statement_block_3
```

Remember indentation rules – you can use spaces or tabs (general convention is 4 spaces or 1 tab). Do not mix types.

Always remember your colon before you enter indentation.

Conditionals: If/elif/else

```
if condition_1:
    statement_block_1
elif condition_2:
    statement_block_2
else:
    statement_block_3
```

Remember indentation rules – you can use spaces or tabs (general convention is 4 spaces or 1 tab). Do not mix types.

Always remember your colon before you enter indentation.

Conditionals: If/elif/else

```
temperature = 73


if temperature > 70:
    print(“Wear shorts.”)
elif temperature > 75:
    print(“Its too hot.”)
```

```
Wear Shorts
```

Conditionals: If/elif/else

```
temperature = 73


if temperature < 70:
    print(“Wear jeans.”)
elif temperature >= 70:
    print(“Its too hot.”)
```

```
Wear Jeans.
```

Conditionals: If/elif/else

```
temperature = 70

if temperature < 70:
    print(“Wear jeans.”)
elif temperature >= 70:
    print(“Its too hot.”)
```

```
Its too hot.
```

Conditionals: If/elif/else

```
#!usr/bin/python3

x = 0

while x <= 10:
        if x == 5:
                print ('midpoint = ', x)
                x+=1
        elif x <= 9:
                print x
                x+=1
        else:
           print(‘end = ’, x)
           x+=1
```

```
0
1
2
3
4
midpoint =  5
6
7
8
end = 9
```

Continue

Continue allows you to skip the remaining steps of the indented block and goto the next loop iteration.

```
a = [0, 1, 2, 3, 4]
for element in a:
    if element == 3:
        continue                #once element is 3, goto next iteration
    print element
```

```
0
1
2
4
```

Continue

Continue allows you to skip the remaining steps of the indented block and goto the next loop iteration.

```
a = [0, 1, 2, 3, 4]
for element in a:
    if element == 3:
        continue                #once element is 3, goto next iteration
    print element
```

```
0
1
2
4
```

Continue

Continue allows you to skip the remaining steps of the indented block and goto the next loop iteration.

```
a = [0, 1, 2, 3, 4]
for element in a:
    if element == 3:
        continue                #once element is 3, goto next iteration
    print element
```

```
0
1
2
4
```

Read and Strip

There are a few ways to open and read lines. This is one of my preferred ways:

```
fh = open(“file.txt")
sequence = ""
for line in fh:
        sequence += line.rstrip('\n')

fh.close
print sequence                    #without header, can add if statement (class 6)
```

```
with open(‘file.txt') as f:
       sequence = [line.rstrip() for line in f]
       sequence = “”.join(sequence[1:])           #if you have header
```

Read and Strip

There are a few ways to open and read lines. This is one of my preferred ways with an if statement.
This allows you to split up your header and sequences into lists:

```
#!/usr/bin/python3

fh = open('example2.fasta')        #open file for reading
lines = fh.readlines()             #array containing each line

header = []                        #list of headers
sequence = []                      #list of sequences
for line in lines:                 #for each line in the lines array
    line = line.rstrip()           #strip line

    if line[0] == '>':             #if line contains ‘>’
        header.append(line)        #add it to the header list
    if line[0] != '>':             #if line doesn’t contain ‘>’
        sequence.append(line)          #add it to the sequence list

print(header)
print(sequence)
```

Conditionals (and, or, not)

```
#!/usr/bin/python3

# get a sequence from the user
DNASeq = input("Enter a DNA Sequence: ")
# Check for length

if len(DNASeq) >= 50 and DNASeq.startswith(“ATCA”) :
# Print a message
    print(”Length greater than 50bp and DNASeq.startswith ATCA”)
```

Conditionals (and, or, not)

```
#!/usr/bin/python3

# get a sequence from the user
DNASeq = input("Enter a DNA Sequence: ")
# Check for length

if len(DNASeq) >= 50 or DNASeq.startswith(“ATCA”) :
# Print a message
    print(”Length greater than 50bp or DNASeq.startswith ATCA”)
```

Conditionals (and, or, not)

```
#!/usr/bin/python3

# get a sequence from the user
DNASeq = input("Enter a DNA Sequence: ")
# Check for length

if len(DNASeq) >= 50 or DNASeq.startswith(“ATCA”) :
# Print a message
    print(”Length greater than 50bp or DNASeq.startswith ATCA”)
```

Programs have to make decisions

```
  for line in chloro.readlines():
    if line.startswith(">"):
```

Line 1 initiates a loop using lines in a file

Line 2 is a conditional test

Programs have to make decisions

There’s not much you can do if your program can’t make decisions

Operators

Comparison operators are part of a larger category of python things called operators

Operators manipulate the value of operands

They do not take arguments

What does this mean?

c = a + b (plus is an operator)

We need operators to build test statements

Arithmetic Operators

Most familiar class of operators

We tried them out back in Week 2

Addition (+), subtraction (-), multiplication (*), division (/), exponentiation (**)

Modulo (%) – really useful for some problems involving loops – divides one number by the other and gives the remainder as the value

Floor division (//) – division in which the digits after the decimal point are removed

Arithmetic Operators

= assigns a value to a variable – that’s the one we all know

Addition (+=), subtraction (-=), multiplication (*=), division (/=), exponentiation (**=) etc

Performs the operation on the two operands, for example adding the variable on the right to the variable on the left, and then follows by assigning the result to the variable on the left

c += a means c = c+a and it would increase as long as you kept, for instance, looping through values of a

Membership and Identity Operators

in/not in (example: if a in mylist:)

is/not is -- values on either side of the operator point to the same value

Comparison Operators

There are a ton of things you can test for with python.

Common ones

equality ==

greater than, less than >, <

greater than/equal to etc >=, <=

not-equality !=

Tests

A test statement places two values on either side of a comparison operator

Evaluates as true if the test is passed, as false if failed

EXAMPLES – let’s try them at the python command line. Make your shell window put on its python suit!

Comparators

Try the following at the python command line:

```
print (3 == 5)
print (3 > 5)
print (3 <=  5)
print (len(“ATGC”) > 5)
print (“GAATTC”.count(“T”) > 1)
print (“ATGCTT”.startswith(“ATG”))
print (“ATGCTT”.endswith(“TTT”))
print (“ATGCTT”.isupper())
print (“ATGCTT”.islower())
print (“V” in “V”, “W”, “L”)
```

Logical operators

AND – both must evaluate True
OR – one or the other must evaluate True
NOT – must evaluate False

True and False are not strings. They are special “truth values” reserved in Python.

Methods that return logic

String methods like .startswith(), .islower() -- they are either true or not true, they evaluate something in the string rather than counting or manipulating

http://www.tutorialspoint.com/python/python_strings.htmlists all the string methods that are available, if you’re curious

Booleans for complex conditions

True and True is True

True and False is False

False and True is False

False and False is False

True or True is True

True or False is True

False or True is True

False or False is False

Not True is False

Not False is True

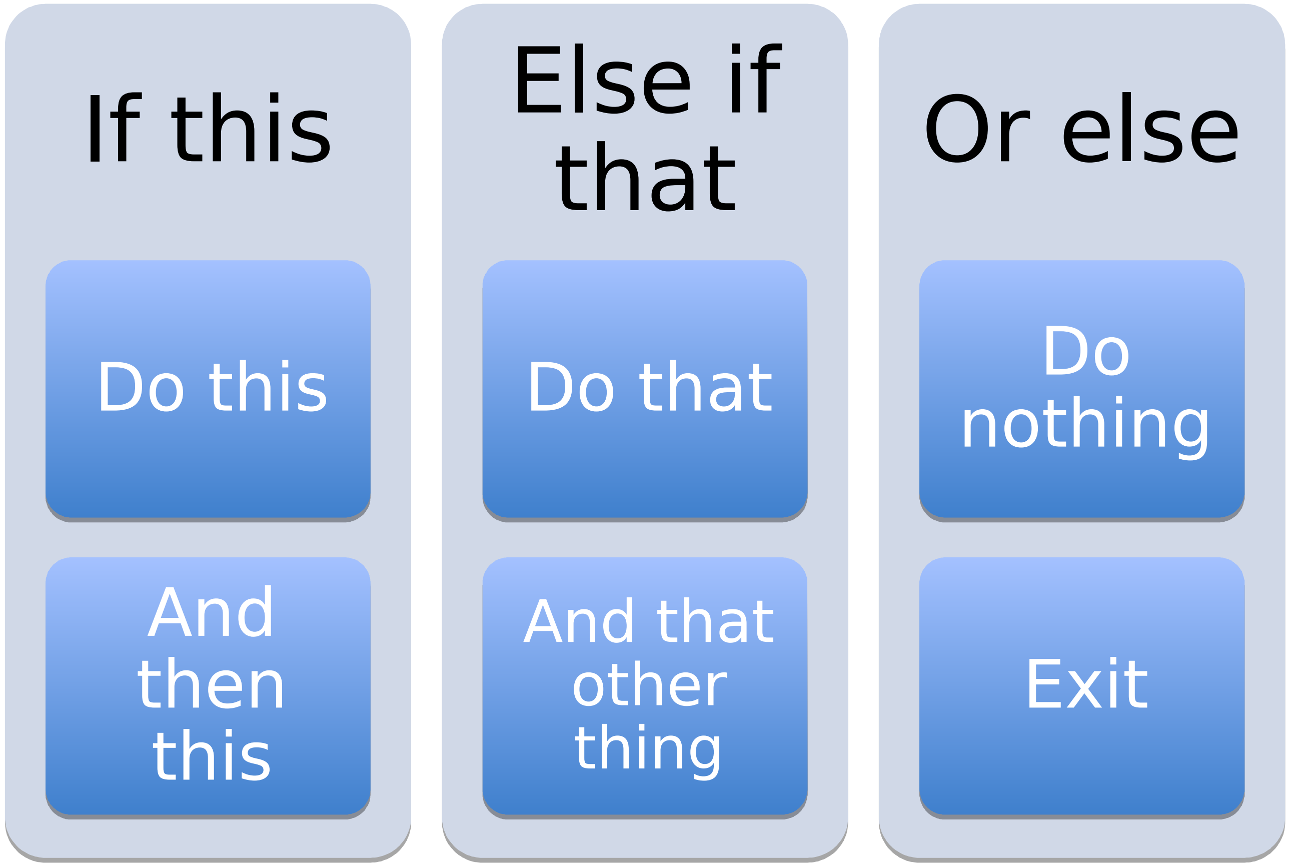
Booleans logic in action

Try out:

```
len("ATGC") == 5 and "ATGC".startswith("C”)
len("ATGC") == 4 or "ATGC".startswith("C”)
len("ATGC") == 4 and "ATGC".startswith(”A”)
len("ATGC") == 5 or "ATGC".startswith("C”)
len("ATGC") == 4 not "ATGC".startswith(”A”)
```

Indentation rules

Logically, code blocks are nested within other blocks



Indentation rules

What is correct indentation?

```
for line in chloro.readlines():
    if line.startswith(">"):
        do a thing…
        and then another thing...
```

Every indentation should be consistent. Examples:

1 tab for each indent level

4 spaces for each indent level

Python code blocks defined by colon & indentation

```
def fib(n):
    print 'n =', n
    if n > 1:
        return n * fib(n - 1)
    else:
        print 'end of the line'
        return 1
```

Indentation rules

If you use spaces, you have to use the same number of spaces consistently – 4 spaces is indent level 1, 8 spaces is indent level 2, etc.

Once the block is done, code continues at the same indentation level of the statement that started the block

Using conditional tests to filter data

Test construct syntax is

```
if:
elif:
else:
```

Filter on length

```
#!/usr/bin/python3

# get a sequence from the user
DNASeq = raw_input("Enter a DNA Sequence: ")
# Check for length
if len(DNASeq) >= 50:
# Print a message
    print ”This one’s a keeper! “
```

Filter on length

```
#!/usr/bin/python3

# get a sequence from the user
DNASeq = raw_input("Enter a DNA Sequence: ")
# Check for length
if len(DNASeq) >= 50:
# Print a message
    print ”This one’s a keeper! “
```

Filter if you find a pattern at the beginning

```
#!/usr/bin/python3

# get a sequence from the user
DNASeq = raw_input("Enter a DNA Sequence: ")
# Check for length
…
if len(DNASeq) >= 50 and float(gcpercent) <= 0.5:
# Print a message
    print ”This one’s a keeper! “

**note, you’d have to calculate this value first
```

Filter if you find the pattern anywhere

```
#!/usr/bin/python3


# get a sequence from the user
DNASeq = raw_input("Enter a DNA Sequence: ")
# get a fragment from the user
fragment = raw_input("Enter a DNA Sequence: ")

# Check for length
…
if len(DNASeq) >= 50 and DNASeq.count[“TACG”] >= 1:
# Print a message
    print ”This one’s a keeper! “

**note, make sure you use an input sequence that has this pattern –
```

Quiz 20

- On canvas now