Conditional statements

Conditional statements allow for selective execution of code depending on the outcome of a comparison. Boolean expressions and conditional statements go hand in hand, as we will see in the exercise.

In [1]:
```
seq = input('Enter a sequence: ')
comp = input('Enter the comp sequence: ')
#p = '|'
print(seq + '\n' + '|'*len(seq) + '\n' + comp)
```

```
Enter a sequence: ATG
Enter the comp sequence: TAC
ATG
|||
TAC
```

Boolean expressions

Boolean expressions are expressions that evaluate to either true or false. They are a common feature of programming languages. The following comparison operators are commonly used in Boolean expressions.

```
  Comparison Operators:
  x == y          # x is equal to y
  x != y          # x is not equal to y
  x > y           # x is greater than y
  x < y           # x is less than y
  x >= y          # x is greater than or equal to y
  x <= y          # x is less than or equal to y
  x is y          # x points to the same object as y (rarely used
)
  x is not y      # x does not point to the same object as y
```

A common error is to use `=` in place of `==`. `=` is an assignment operator (i.e. set x = y), whereas `==` is a comparison operator (i.e. is x = y).

Assign a volume to the variable `vol`:

In [2]:
```
vol = 100
```

What type of value is the variable? Confirm your answer using the `type()` function:

In [5]:
```
type(vol)
```

Out[5]:
```
int
```

When evaulated by the Python interpreter, Boolean expressions that evaulate produce special values - `True` if true and `False` if false. `True` and `False` belong to a class of values called `bool` .

In [6]:
```
type(False)
```

Out[6]:
```
bool
```

Use the `==` operator to test if `vol` is equal to 100:

In [7]:
```
vol == 100
```

Out[7]:
```
True
```

Use the `<=` operator to determine if `vol` is less than or equal to 100:

In [10]:
```
vol <= 100
```

Out[10]:
```
True
```

Use the `!=` operator to determine if `vol` is not equal to 100:

In [11]:
```
vol != 100
```

Out[11]:
```
False
```

Assign an RNA sequence to two different variables, for example `rna1` and `rna2` :

In [12]:
```
rna1 = 'AUGU'
rna2 = 'AUGU'
```

Compare the two RNA sequences to determine if they are equivelent using `==` :

In [13]:
```
rna1 == rna2
```

Out[13]:
```
True
```

Compare the two RNA sequence variable from above using the `is` operator:

In [14]:
```
rna1 is rna2
```

Out[14]:
```
True
```

The `is` operator at first glance seems similar to `==` but it is quite different. `is` tests whether two values point to the same object. The variables `rna1` and `rna2` point to the same object, a value that python has interned or stored away. But it's dangerous to use `is` when making comparisons unless you know the object has been interned. Let's look at an example using a type of object we haven't seen yet, a list:

In [15]:
```
a = [1,2]
b = a
b is a
```

Out[15]:
```
True
```

In [16]:
```
a = [1,2]
b = [1,2]
b == a
```

Out[16]:
```
True
```

In [17]:
```
a = [1,2]
b = [1,2]
b is a
```

Out[17]:
```
False
```

As a rule of thumb use `==` to make comparisons (unless you're asking if an object `is None` as we'll seet later in the course).

Logical operators

There are three logical operators in Python, `and` , `or` , and `not` . These operators are often represented by symbols in other programming languages. A nice feature of Python is that it uses the English words for these operators.

Assign a number to a variable, such as `n` :

In [19]:
```
n = 100
```

Test if the number is greater than 95 but less than 105:

In [22]:
```
n > 95 and n < 105
```

Out[22]:
```
True
```

The expressions on either side of `and` are evaulated independently, thus we can't write the comparison this way:

In [23]:
```
n > 95 and < 105
```

```
  File "<ipython-input-23-230e87afb517>", line 1
    n > 95 and < 105
                 ^
SyntaxError: invalid syntax
```

The code above generates errors because `< 105` by itself is not an expression that can be evalauted.

Test if `n` is equal to 100 or 101:

In [25]:
```
n == 100 or n == 101
```

Out[25]:
```
True
```

Test if `n` is not equal to 100 or 101 using the `not` operator:

In [28]:
```
not(n == 100 or n == 101)
```

Out[28]:
```
False
```

As you can probably see, negating something with the `not` operator often makes code difficult to follow so when possible it should be avoided.

Test if `n` is not eqaul to 100 or 101 using the `!=` operator:

In [30]:
```
n != 100 and n != 101
```

Out[30]:
```
False
```

Chained conditionals

What if we want to test multiple conditions or execute distinct blocks of code depending on which of multiple possible conditions is true? We can simply add another condition, *else if*, to an `if` statement (previewed on Tuesday). Python uses the shorthand `elif` in its syntax for *else if* (it saves a couple keystrokes). Notice that conditional statements use boolean expressions:

In [ ]:
```
if boolean expression evaluates to True:
    block of code
elif alternative boolean expression evaluates to True:
    block of code
else: # (i.e. neither boolean expression evaluates to True)
    block of code
```

Prompt the user for the mass of an RNA sample:

In [34]:
```
mass = float(input('Enter mass of RNA: '))
```

```
Enter mass of RNA: 5
```

Next, prompt the user and the units of mass (e.g. ng or ug):

In [35]:
```
units = input('Enter ng or ug: ')
```

```
Enter ng or ug: ug
```

Determine if the mass is less than 10 ng, assuming a mass less than 10 ng is insufficient for the experiment:

In [36]:
```
if mass < 10 and units == 'ng':
    print('Insufficient RNA for experiment')
elif mass < .01 and units == 'ug':
    print('Insufficient RNA for experiment')
else:
    print('Proceed with experiment')
```

```
Proceed with experiment
```

It is not actually necessary to include an `elif` statement in the above example if we use the logical operator `or` :

In [ ]:
```
if (mass < 10 and units == 'ng') or (mass <0.01 and units == 'ug'):
    print('Insufficient RNA for experiment')
else:
    print('Proceed with experiment')
```

Order of presedence: In Python, `and` is higher precedence than `or`. The comparison operators are all higher precedence than `and` and `or`. Thus, parentheses are often optional but will usually make your code easier to read.

Exercise 3a

Compute whether a 3-nt sequence input by the user is start codon (AUG), a stop codon (UAG, UAA, or UGA), or neither.

In [39]:
```
seq = input('Insert 3 nt seq: ')
if seq == 'AUG':
    print('This is a start codon')
elif seq == 'UAG' or seq == 'UAA' or seq == 'UGA':
    print('This is stop codon')
else:
    print('This seq is neither a stop or start codon')
```

```
Insert 3 nt seq: AAA
This seq is neither a stop or start codon
```

Nested conditionals

It is occasionally necessary, or at least more convienent, to nest conditionals within conditionals:

In [ ]:
```
if primary expression evaluates to True:
    if secondary expression evaluates to True:
        block of code
    else:
        block of code
elif alternative primary expression evaluates to True:
    block of code
else:
    block of code
```

Exercise 3b

Modify your code from exercise 3a so that the user is prompted to specificy whether they entered DNA and modify the code to work on both DNA and RNA using a nested conditional.

In [42]:
```
seq = input('Enter a 3 nt seq: ')
dna = input('DNA (y/n): ')
if dna == 'y':
    if seq == 'ATG':
        print('This is a start codon')
    elif seq == 'TAG' or seq == 'TAA' or seq == 'TGA':
        print('This is stop codon')
    else:
        print('This seq is neither a stop or start codon')
elif dna == 'n':
    if seq == 'AUG':
        print('This is a start codon')
    elif seq == 'UAG' or seq == 'UAA' or seq == 'UGA':
        print('This is stop codon')
    else:
        print('This seq is neither a stop or start codon')
else:
    print('Sequence of unknown type entered')
```

```
Enter a 3 nt seq: ATG
DNA (y/n): d
Sequence of unknown type entered
```

`try` and `except`

An error detected during execution of code is called an exception. The error message is often ugly and difficult to understand unless you're familiar with Python:

In [43]:
```
n = 'ATG'
print(n**2)
print('Program complete.')
```

```
---------------------------------------
TypeErrorTraceback (most recent call last)
<ipython-input-43-22cab474cc0e> in <module>()
      1 n = 'ATG'
----> 2 print(n**2)
      3 print('Program complete.')

TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int
'
```

`try` and `except` provide a convienent way to 'catch an exception' and provide a more informative custom error message. The syntax is as follows:

In [ ]:
```
try:
    block of code
except:
    block of code (usually prints a user friendly error message or return
```

If the code in the `try` clause is executable without generating an error, than the code in the `except` clause, which typically prints a user defined error message is ignored. If an error is generated, than the `except` clause is executed. `try` and `except` can be used to detect different types of exceptions and execute different blocks of code depending on the error.

Assign a sequence to a variable, `n` , and then attempt to calculate the square of `n` within a `try` clause. Throw an exeption using `except` if the calculation is invalid:

In [53]:
```
n = 'ATG'
try:
    n**2
except:
    print('n must be a valid number')
print('Program complete.')
```

```
n must be a valid number
Program complete.
```

An additional advantage to the `try` clause is that code outside of it is still executed even if the code within the `try` clause throws an exception.

Membership operators

The `in` and `not in` operators provide a very simple way of determining if something is in something else, such as whether a string is in a variable.

Assign a DNA sequence to a variable:

In [54]:
```
seq = 'ATGTGTGA'
```

Test if the sequence contains the string ATG:

In [57]:
```
'ATG' in seq
```

Out[57]:
```
False
```

Test if the sequence does not contain the string TGA:

In [60]:
```
'TGA' not in seq
```

Out[60]:
```
False
```

The `len()` function

The length function, `len()`, returns the length of an object. The object can be many different things, but when used on a string `len()` returns the number of characters contained in the string.

Assign a sequence to a variable `seq` and identify its length:

In [64]:
```
seq = 'ATGTAC'
len(seq)
```

Out[64]:
```
6
```

Of course we can store the value returned by `len()` as a variable:

In [1]:
```
seq = 'ATG'
length = len(seq)
len(seq)
```

Out[1]:
```
3
```

Exercise 3c

Write a script that prompts a user for two sequences and does the following:

1) Computes whether or not each sequence is DNA (contains T) or RNA (contains U).

2) If both DNA and RNA sequences are entered, prints an error message.

3) Concatenates the sequences and calculates the length of the concatenated sequence.

Questions

Q. What are Boolean expressions?

A.

Q. What are the three logical operators in Python?

A.

Q. What are the membership operators in Python and how are they used?

A.

Additional exercises

Except where noted otherwise, you can use whatever method you prefer to do the exercises (notebook, shell, script).

3d) Using the Python interpreter shell, assign a decimal value to a variable. Test if the variable is less than 10 or greater than 10. Remember to indent in the interpretor.

3e) Write a script that prompts the user for a number and prints the inverse of the number. Include a `try` and `except` that catches the exception if 0 is entered.

3f) Python evaluates expressions from left to right and stops evaluating a statement when it is no longer necessary. For example:

In [ ]:
```
x = 5
y = 0
if x < y and x/y < 3:
    print ('x is less than y and x/y is less than 3')
```

Once `x < y` evaluates as `False`, there is no need for Python to evaluate the rest of the statement. This is referred to as *short-circuiting* the evaulation. This feature can be exploited by introducing a *guardian pattern* in which an additional evaluation, called a *guardian pattern* is introduced that prevents a comparison from being made that would generate an error and disrupt the program. The following code will generate an error message because it is not possible to divide a number by 0:

In [ ]:
```
x = 5
y = 0
if x > y and x/y < 3:
    print ('x is greater than y and x/y is less than 3')
```

Modify the code above with a *guardian pattern* (i.e. an additional expression that guards against an illegal division) within the `if` statement, without modifying either of the two existing expressions.

3g) Compute how many additional points a student needs to obtain an A in a course (assume 90 is an A). Your code should prompt the user for their current score out of 100 pts possible. If the score entered does not fall between 0-100, give the user one more opportunity to enter a score within the permissible range.

In [ ]:

3h) Modify the code from 3g so that it does not exceed 9 lines.