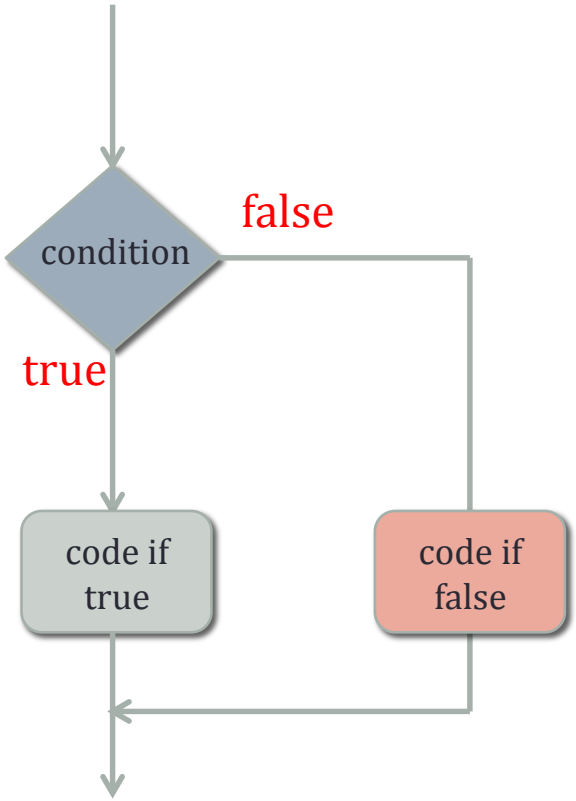
IFS AND LOOPS

PYTHON FOR GENOMIC DATA SCIENCE

Decision Making



if *condition*

block of code to execute if condition is true

else

other block of code to execute if condition is false

The `if` Statement

Take a look at the following code:

```
>>> dna=input('Enter DNA sequence:')
Enter DNA sequence:agcgcgggtatatatatgcnccann


>>> if 'n' in dna :

        nbases=dna.count('n')

        print("dna sequence has %d undefined bases " % nbases)



dna sequence has 3 undefined bases
```

The `if` Statement

Take a look at the following code:

```
>>> dna=input('Enter DNA sequence:')
Enter DNA sequence:agcgcgggtatatatatgcnccann
>>> if 'n' in dna :
        nbases=dna.count('n')

        print("dna sequence has %d undefined bases " % nbases)

dna sequence has 3 undefined bases
```

condition

block of code to execute if condition is true

notice the indentation that defines the block

Boolean Expressions

- The *condition* in the `if` statement is called a *Boolean expression*.
- *Boolean expressions* are *expressions* that are either *true* or *false*.

```
>>> 0<1
True
>>> len('atgcgt')>=10
False
```

- Boolean expressions are formed with the help of *comparison*, *identity*, and *membership operators*.

Comparison Operators

Comparison	Operator
Equal	==
Not equal	!=
Less than	<
Greater than	>
Less than or equal to	<=
Greater than or equal to	>=

```
>>> 'a'=='A'
False
>>> 'GT' != 'AG'
True
```

Do not confuse the assignment sign = with the comparison operator == !

Comparison Operators

Comparison	Operator
Equal	==
Not equal	!=
Less than	<
Greater than	>
Less than or equal to	<=
Greater than or equal to	>=

```
>>> 'a'=='A'
False
>>> 'GT' != 'AG'
True
>>> 'A'<'C'
True
>>> 10+1==11
True
```

Do not confuse the assignment sign = with the comparison operator == !

Membership Operators

Operator	Description
in	True if it finds a variable in the specified sequence and false otherwise.
not in	True if it does not finds a variable in the specified sequence and false otherwise.

```
>>> motif='gtccc'
>>> dna='atatattgtcccattt'
>>> motif in dna
True
```

Identity Operators

Operator	Description
is	True if the variables on either side of the operator point to the same object and false otherwise.
is not	False if the variables on either side of the operator point to the same object and true otherwise.

Identity Operators

Operator	Description
is	True if the variables on either side of the operator point to the same object and false otherwise.
is not	False if the variables on either side of the operator point to the same object and true otherwise.

```
>>> alphabet=['a','c','g','t']
>>> newalphabet=alphabet[:]
>>> alphabet == newalphabet
True
>>> alphabet is newalphabet
False
```

Alternative Execution

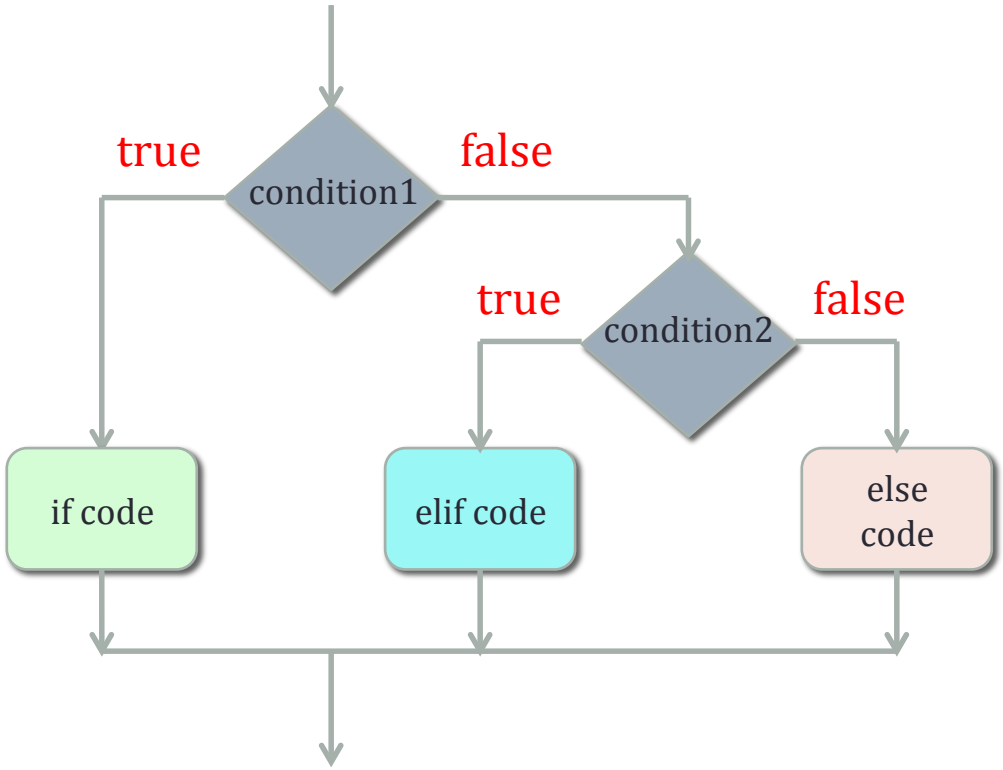
check_dna.py

```
#!/usr/bin/python

dna=input('Enter DNA sequence:')

if 'n' in dna :
    nbases=dna.count('n')
    print("dna sequence has %d undefined bases " % nbases)

else:
    print("dna sequence has no undefined bases")
```

Multiple Alternative Executions

Sometimes it is convenient to test several conditions in one if structure.

```
if condition 1:
    do action 1
elif condition 2:
    do action 2
elif condition 3:
    do action 3
...
elif condition n:
    do action n
else:  # none of conditions 1,2,…,n are true
    do something else
```



Multiple Alternative Executions (cont'd)

Sometimes it is convenient to test several conditions in one if structure.

check_dna.py

```
#!/usr/bin/python

dna=input('Enter DNA sequence:')
if 'n' in dna :
    print("dna sequence has undefined bases ")
elif 'N' in dna :
    print("dna sequence has undefined bases ")
else:
    print("dna sequence has no undefined bases")
```

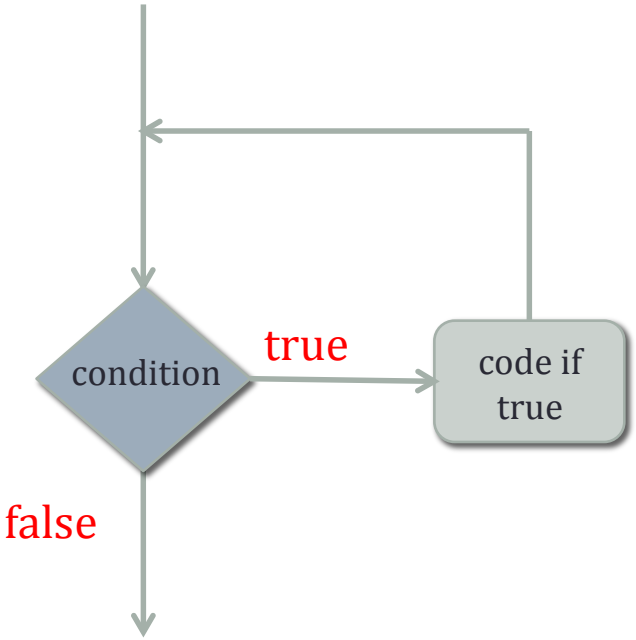
Logical Operators

```
and    - True if both conditions are true
or     - True if at least one condition is true
not    - True if condition is false
```

check_dna.py

```
#!/usr/bin/python

dna=input('Enter DNA sequence:')
if 'n' in dna or 'N' in dna:
    nbases=dna.count('n')+dna.count('N')
    print("dna sequence has %d undefined bases " % nbases)
else:
    print("dna sequence has no undefined bases")
```

Loops



while *condition*

block of code to execute while condition is true

The `while` Loop

Problem. Given a DNA sequence find the positions of all canonical donor splice site candidates in the sequence.

find_donor.py

```
#!/usr/bin/python

dna=input('Enter DNA sequence:')
pos=dna.find('gt',0) # position of donor splice site
```

The `while` Loop

Problem. Given a DNA sequence find the positions of all canonical donor splice site candidates in the sequence.

find_donor.py

```
#!/usr/bin/python

dna=input('Enter DNA sequence:')
pos=dna.find('gt',0) # position of donor splice site
while pos>-1 :
    print("Donor splice site candidate at position %d"%pos)
    pos=dna.find('gt',pos+1)
```

condition

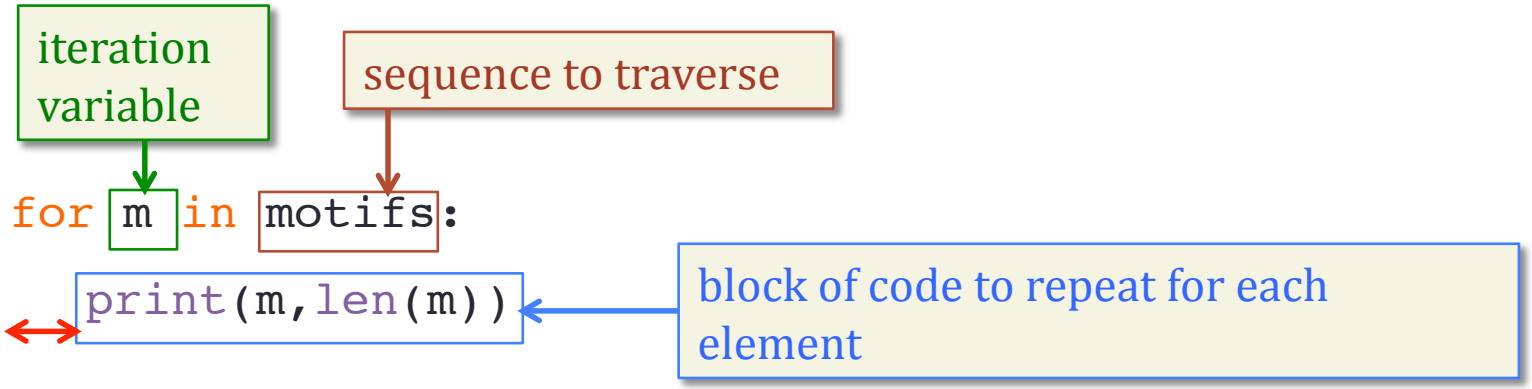
block of code to execute while condition is true

notice the indentation that defines the while block

The `for` Loop

Python's `for` loop iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence.

```
>>> motifs=["attccgt","agggggtttttcg","gtagc"]
>>> for m in motifs:
...     print(m,len(m))
```




```
attccgt 7
agggggtttttcg 13
gtagc 5
```

The `range()` Function

The `range()` built-in function allows you to iterate over a sequence of numbers:

```
>>> for i in range(4):
...     print(i)
0
1
2
3
```


```
>> for i in range(1,10,2):
...     print(i)
1
3
5
7
9
```

The `range( )` Function (cont'd)

Problem. Find if all characters in a given protein sequence are valid amino acids.

Pseudocode

```
for each character in protein sequence :
    if character is not amino acid :
        print invalid character and its position in protein
```

```
>>> protein='SDVIHRYKUUPAKSHGWYVCJRSRFTWMVWWRFRSCRA'
>>> for i in range(len(protein)):
...     if protein[i] not in 'ABCDEFGHIKLMNPQRSTVWXYZ':
...         print("protein contains invalid amino acid %s at
position %d"%(protein[i],i))

protein contains invalid amino acid U at position 8
protein contains invalid amino acid U at position 9
protein contains invalid amino acid J at position 20
```

Breaking Out Of Loops

Problem. Suppose we are only interested in finding if a protein sequence is valid, not where are all the invalid characters in the sequence.

invalid amino acid character

```
>>> protein='SDVIHRYKUUPAKSHGWYVCJRSRFTWMVWWRFRSCRA'
>>> for i in range(len(protein)):
...     if protein[i] not in 'ABCDEFGHIKLMNPQRSTVWXYZ':
...         print("this is not a valid protein sequence!")
...         break
```

the break statement terminates the nearest enclosing loop (a `for` or `while` loop)

```
this is not a valid protein sequence!
```

The `continue` Statement

The `continue` statement causes the program to continue with the next iteration of the nearest enclosing loop, skipping the rest of the code in the loop.

Problem. Delete all invalid amino acid characters from a protein sequence.

```
>>> protein='SDVIHRYKUUPAKSHGWYVCJRSRFTWMVWWRFRSCRA'
>>> corrected_protein=''
```

The `continue` Statement

The `continue` statement causes the program to continue with the next iteration of the nearest enclosing loop, skipping the rest of the code in the loop.

Problem. Delete all invalid amino acid characters from a protein sequence.

```
>>> protein='SDVIHRYKUUPAKSHGWYVCJRSRFTWMVWWRFRSCRA'
>>> corrected_protein=''
>>> for i in range(len(protein)):
...     if protein[i] not in 'ABCDEFGHIKLMNPQRSTVWXYZ':
...             continue
...     corrected_protein=corrected_protein+protein[i]
...
>>> print("Corrected protein sequence is:%s"%corrected_protein)

Corrected protein sequence is:SDVIHRYKPAKSHGWYVCRSRFTWMVWWRFRSCRA
```

The `continue` Statement (cont'd)

Using the continue statement improves the readability of your code:

```
for i in range(n):
    if  condition_1:
        function_1(i)
        if condition_2:
            funtion_2(i)
            if condition_3:
                function_3(i)
                ...
```

The `continue` Statement (cont'd)

Using the continue statement improves the readability of your code:

```
for i in range(n):
    if  condition_1:
        function_1(i)
        if condition_2:
            funtion_2(i)
            if condition_3:
                function_3(i)
                ...
```

```
for i in range(n):
    if  not condition_1:
        continue
    function_1(i)
    if not condition_2:
        continue
    funtion_2(i)
    if not condition_3:
        continue
    function_3(i)
    ...
```

The `else` Statement Used With Loops

Loop statements may have an `else` clause.

- If used with a `for` loop, the `else` statement is executed when the loop has exhausted iterating the list.
- If used with a `while` loop, the `else` statement is executed when the condition becomes false.

The `else` statement is not executed if the loop is terminated by the `break` statement!

An Example Of Using `else` With A `for` Loop

Problem. Find all prime numbers smaller than a given integer.

```
>>> N=10
>>> for y in range(2, N):
...     for x in range(2, y):
...         if y % x == 0:
...             print(y, 'equals', x, '*', y//x)
...             break
```

```
9 equals 3 * 3
```

Solution adapted from https://docs.python.org/3/tutorial/controlflow.html

An Example Of Using `else` With A `for` Loop

Problem. Find all prime numbers smaller than a given integer.

```
>>> N=10
>>> for y in range(2, N):
...     for x in range(2, y):
...         if y % x == 0:
...             print(y, 'equals', x, '*', y//x)
...             break
...     else:
...         # loop fell through without finding a factor
...          print(y, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

Solution adapted from https://docs.python.org/3/tutorial/controlflow.html

The `pass` Statement

- Python's `pass` statement is a placeholder: it does nothing.
- It is used when a statement is required syntactically but you do not want any command or code to execute:

```
>>> if motif not in dna:
...     pass
... else:
...     some_function_here(motif,dna)
```

The `pass` Statement

- Python's `pass` statement is a placeholder: it does nothing.
- It is used when a statement is required syntactically but you do not want any command or code to execute:

```
>>> if motif not in dna:
...     pass
... else:
...     some_function_here(motif,dna)
```

- Sometimes you can use the `pass` statement also when you didn't yet write the code for a particular situation but you need the placeholder so that the syntax of the rest of your program is correct.