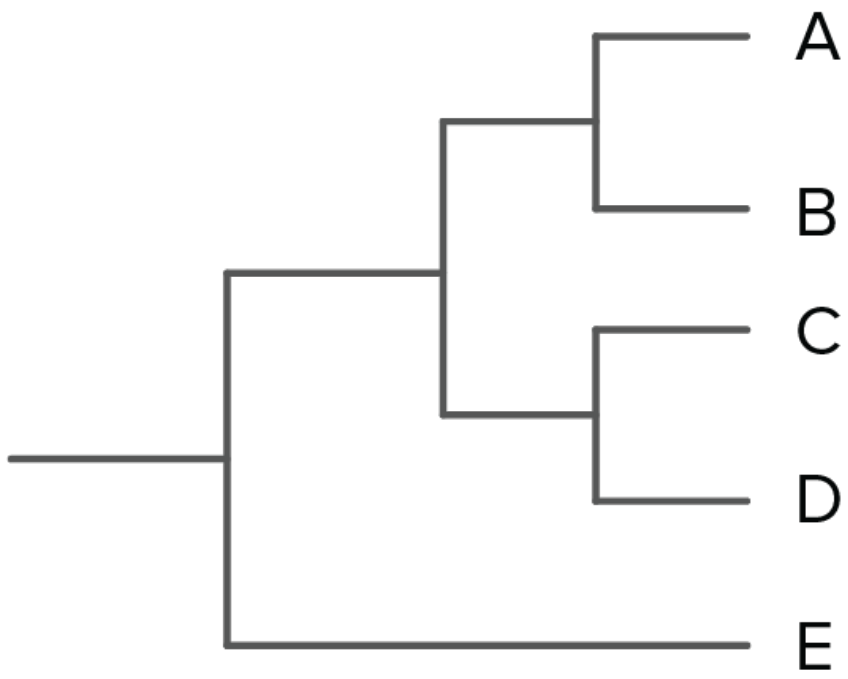
Quiz Section Week 4
April 18, 2017

Finish Fitch algorithm practice

Dictionaries, For loops, Functions

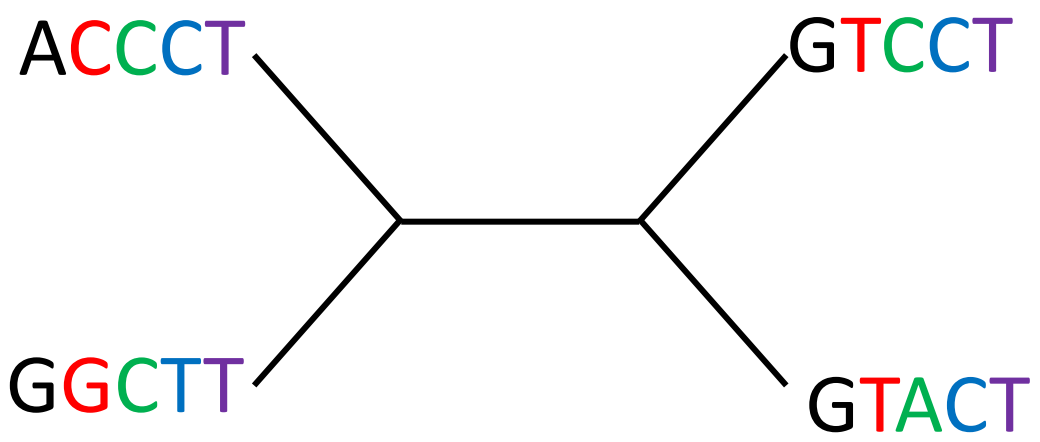
Fitch algorithm: What are we doing?

- The *small* parsimony problem
- Analyzing a *single* tree
  - Min changes required (parsimony score)
  - Parsimonious assignment of internal node traits



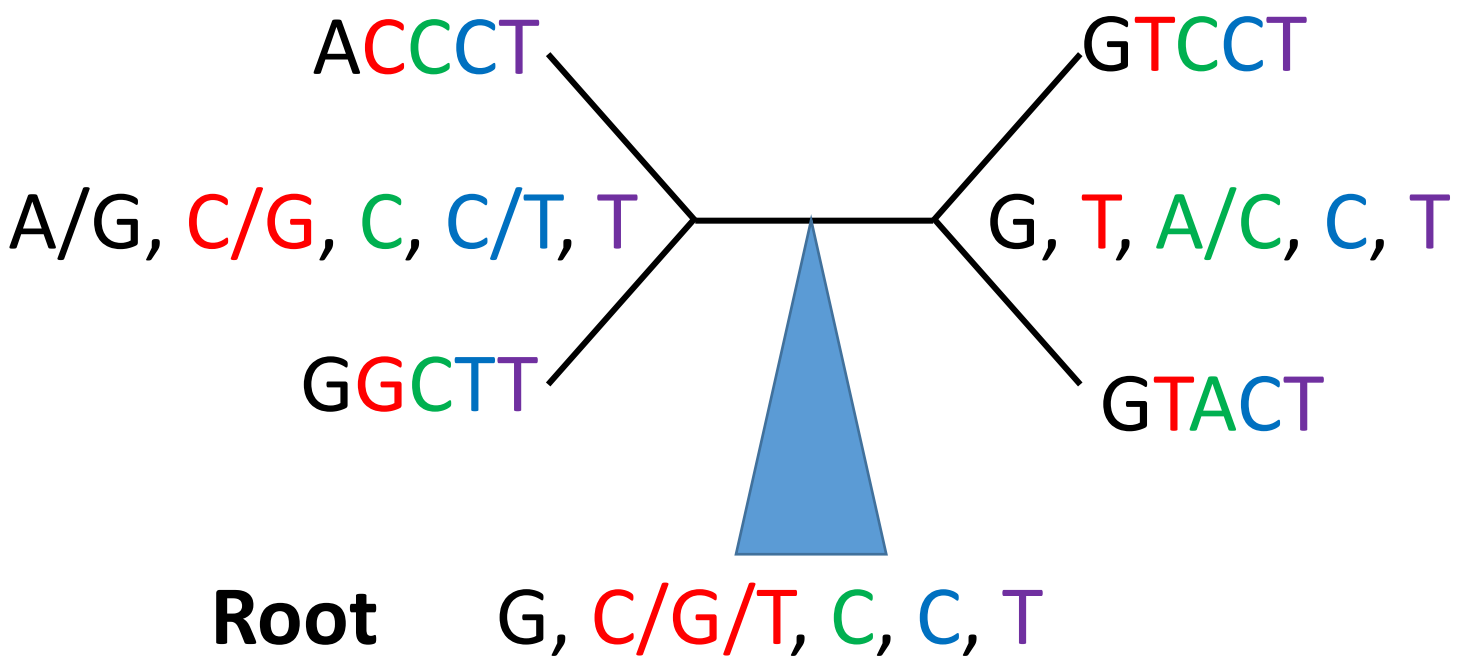
Fitch algorithm practice: bottom-up phase

Goal: Assign possible values to internal nodes, calculate parsimony score



Fitch algorithm practice: bottom-up phase

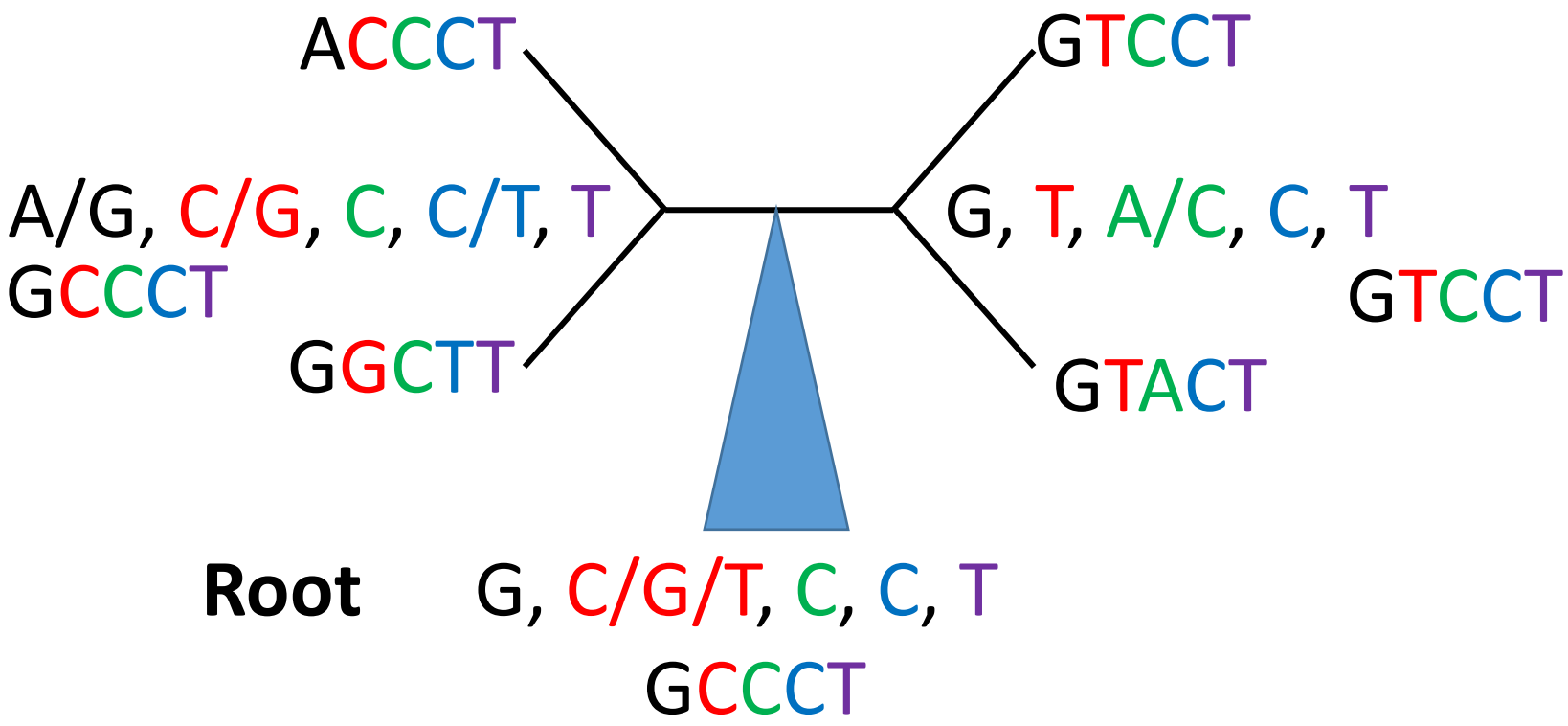
Goal: Assign possible values to internal nodes, calculate parsimony score



Fitch algorithm practice: top-down phase

Goal: Pick a single consistent set of values for internal nodes

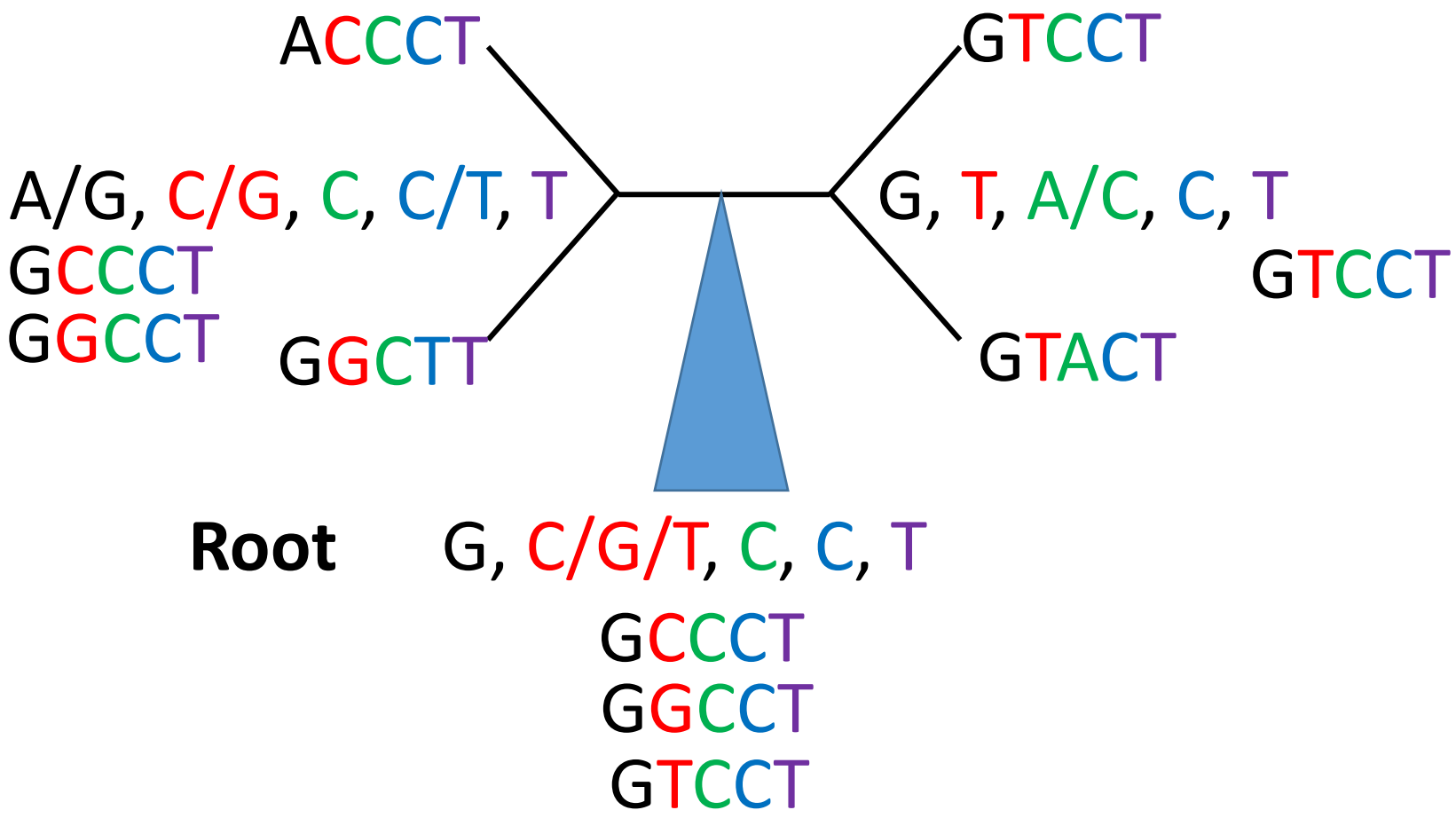
Intuition: if possible, assign same state as parent



Fitch algorithm practice: top-down phase

Goal: Pick a single consistent set of values for internal nodes

Intuition: if possible, assign same state as parent



Programming

What's a function?

Function: reusable pieces of code, that take zero or more arguments, perform some actions, and return one or more values

e.g the function **len**

```
>>> len("AGCAGTTTT")
9
```

- arguments: a string or list
- actions: count the number of characters or elements
- return: the integer length of the string or list

How about the function **range**?

```
>>> range(1,4)
[1,2,3]
```

- arguments:
- actions:
- return:

Methods are defined functions that are applied to a specific variable of a given type

String methods: We use the "." to be able to access and apply them to a particular string

```
>>> s = "GATTACA"
>>> s.find("ATT")
1
>>> s.count("T")
2
>>> s.lower()
'gattaca'
>>> s+s
'GATTACAGATTACA'
```

```
>>> s.upper()
'GATTACA'
>>> s.replace("G", "U")
'UATTACA'
>>> s.replace("C", "U")
'GATTAUA'
>>> s.replace("AT",
"**")
'G**TACA'
>>> s = "GAT TAC CAT"
>>> s.split()
['GAT','TAC','CAT']
```

Another data type: Dictionaries

- a data structure that consists of an unordered set of *key: value* pairs
  - think of as *word: definition* pairs!

Q: How could we encode the entire genetic code?

https://docs.python.org/3/tutorial/datastructures.html

Dictionaries: How could we encode the entire genetic code?

```
>>> genetic_code = {"ATG": "Start", "TGA": "Stop", "TAG":
"Stop"}
>>> genetic_code["TAA"] = "Stop"
>>> genetic_code.get("TGA")
'Stop'
>>> genetic_code["TGA"]
'Stop'
>>> genetic_code.get("sss") #nothing or 'None' if not defined
>>> genetic_code["sss"]
KeyError: 'ttt'
```

https://docs.python.org/3/tutorial/datastructures.html

Some useful dictionary methods

```
>>> genetic_code.items()
[('TAA', 'Stop'), ('TGA', 'Stop'), ('TAG', 'Stop'),
('ATG', 'Start')]
>>> genetic_code.keys()
['TAA', 'TGA', 'TAG', 'ATG']
>>> genetic_code.values()
['Stop', 'Stop', 'Stop', 'Start']
```

Another use of dictionaries: store counts of named elements

Example: Calculate # of each nucleotide in a sequence

```
sequence = "GACCCT"
nuc_counts = {'A': 0, 'C': 0, 'T':0, 'G': 0}
for nuc in sequence:
     #Add to the count for the given nucleotide
```

Another common use of dictionaries: store counts of named elements

Calculate # of each nucleotide in a sequence

```
sequence = "GACCCT"
nuc_counts = {'A': 0, 'C': 0, 'T':0, 'G': 0}
for nuc in sequence:
     nuc_counts[nuc] = nuc_counts[nuc] + 1
```

More on For loops

Example: List all possible codons

```
all_codons = []
for nuc in "ACTG":
    for nuc2 in "ACTG":
        for nuc3 in "ACTG":
            codon = nuc+nuc2+nuc3
            all_codons.append(codon)
print all_codons
```

Nested loops!

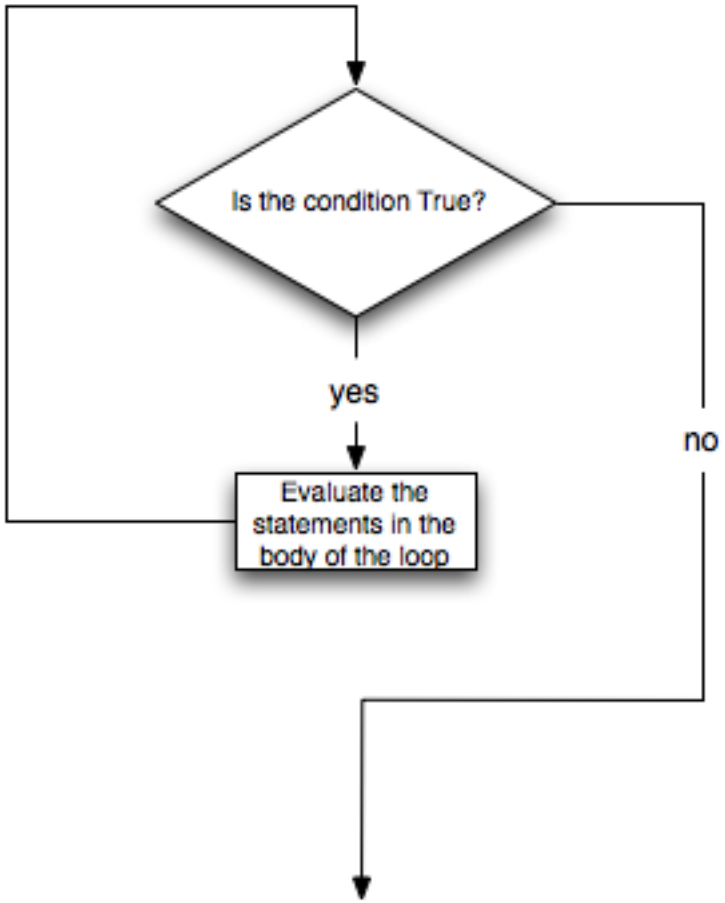
Output? How many codons?

Breaking out of a for loop

Print codons 1 at a time until we hit any stop codon, then stop

```
print(all_codons)
genetic_code = {"ATG": "Start", "TGA": "Stop",
"TAG": "Stop"}
for codon in all_codons:
    print(codon)
    if genetic_code.get(codon) == 'Stop':
        break
```

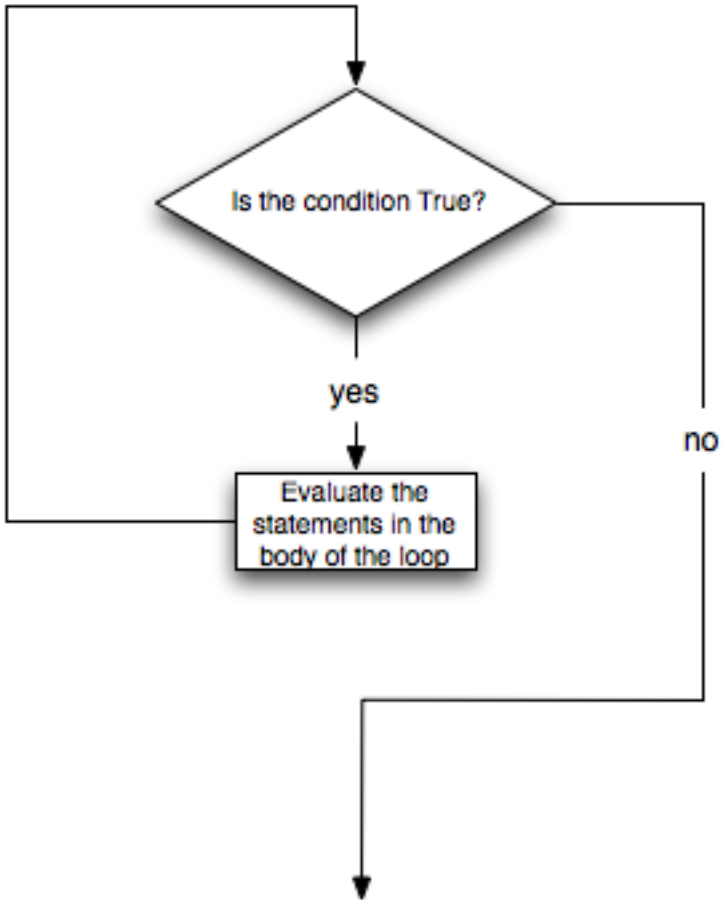
While loops: another option when you don't know how many repeats you need to do




```
counter = 0
aa = ''

while aa != 'Stop':
    codon = all_codons[counter]
    aa = genetic_code.get(codon)
    print(aa)
    counter = counter + 1
```

While loops can go wrong easily




```
counter = 0
aa = ''
codon = all_codons[counter]

while aa != 'Stop':
    aa = genetic_code.get(codon)
    print(aa)
    counter = counter + 1
```

Often, inside of a loop we want to apply a function!

Very common program structure:

```
all_results = []
for element in data:
    #Calculate something from each element in a dataset
    result = do_something(element)
    #Compile all the calculation results in a list
    all_results.append(result)
```

Writing your own functions

argument(s)

```
def do_something(datapoint):
    #Whatever your calculation is
    result = datapoint*100
    return result
```

action (s)

output returned

Why write our own functions?

- Avoid repetition, use the same piece of code in different ways

- Better organized, easier-to-understand code
  - harder to make mistakes, easier to find them

Write a function that transcribes DNA sequence into RNA sequence

```
def transcribe(dna_sequence):
```

Write a function that transcribes DNA sequence into RNA sequence

```
def transcribe(dna_sequence):
    rna_sequence = dna_sequence.replace('T','U')
    return rna_sequence
```

Using your function

```
def transcribe(dna_sequence):
    rna_sequence = dna_sequence.replace('T','U')
    return rna_sequence

sequence = "ATTGCCT"
print(transcribe(sequence))
print(rna_sequence)
```

Using your function

```
def transcribe(dna_sequence):
    rna_sequence = dna_sequence.replace('T','U')
    return rna_sequence

sequence = "ATTGCCT"
print(transcribe(sequence))
print(rna_sequence)
```

Not defined outside of function!!