Loops & Iteration

Learning Objects

1. Looping
- Understand the purpose and types of loops (`for`, `while`).
- Write basic loop constructs to repeat actions.

2. Indefinite and Infinite Loops
- Distinguish between indefinite loops and infinite loops.
- Implement an indefinite loop for user input.

3. Loop Forever
- Define what it means for a loop to run indefinitely.
- Create an example of an infinite loop and discuss its uses.

4. Breaking Out of a Loop
- Learn to use the `break` statement to exit a loop.
- Write code that demonstrates the use of `break` in loops.

5. iteration.
- Implement examples showing its effect on loop flow.

6. Iteration with Range Function
- Use the `range()` function to generate sequences of numbers.
- Implement loops that iterate a specified number of times.

7. Iteration with Index
- Understand indexing in collections.
- Write examples that access elements by index during iteration.

Indefinite Loops

- `while` loops are called "indefinite loops" because they keep going until a logical condition becomes False
- The loops we'll see next are pretty easy to examine to see if they will terminate or if they will be "infinite loops"
- Sometimes it is a little harder to be sure if a loop will terminate, or what happens inside a loop with assignment, counters, incrementor, data structures

Loop / Loop with Final Clause

The basic form of loop begins with the keyword **`while`** and an expression.

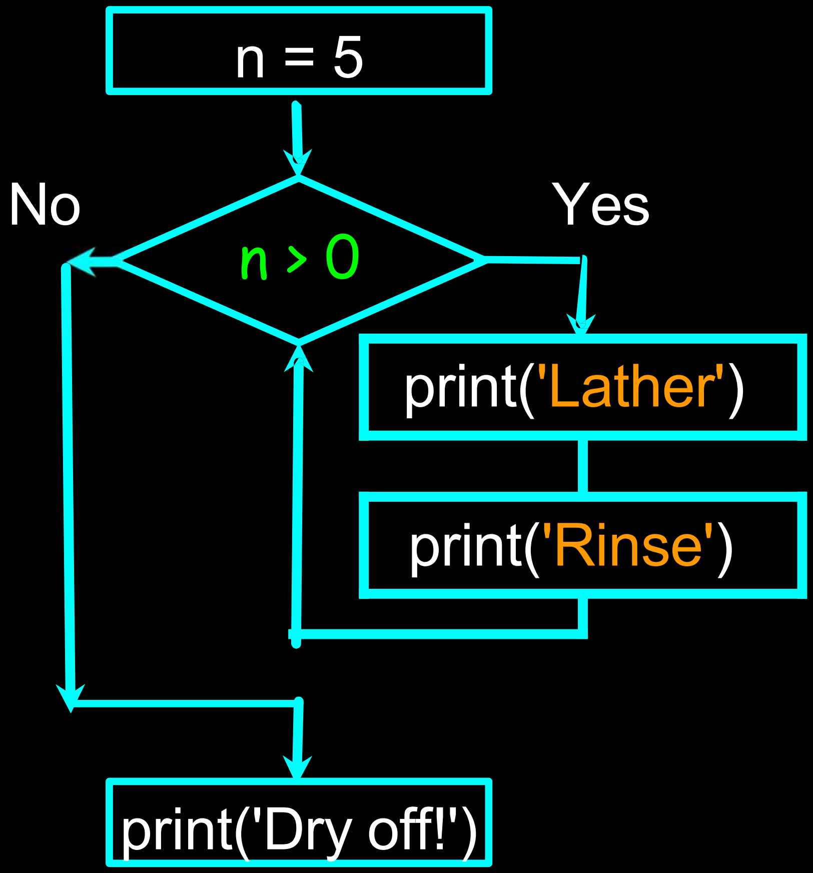
`while` *expression:*
    statements

This form of loop statement adds an else clause whose statements are executed after the expression evaluates to false.

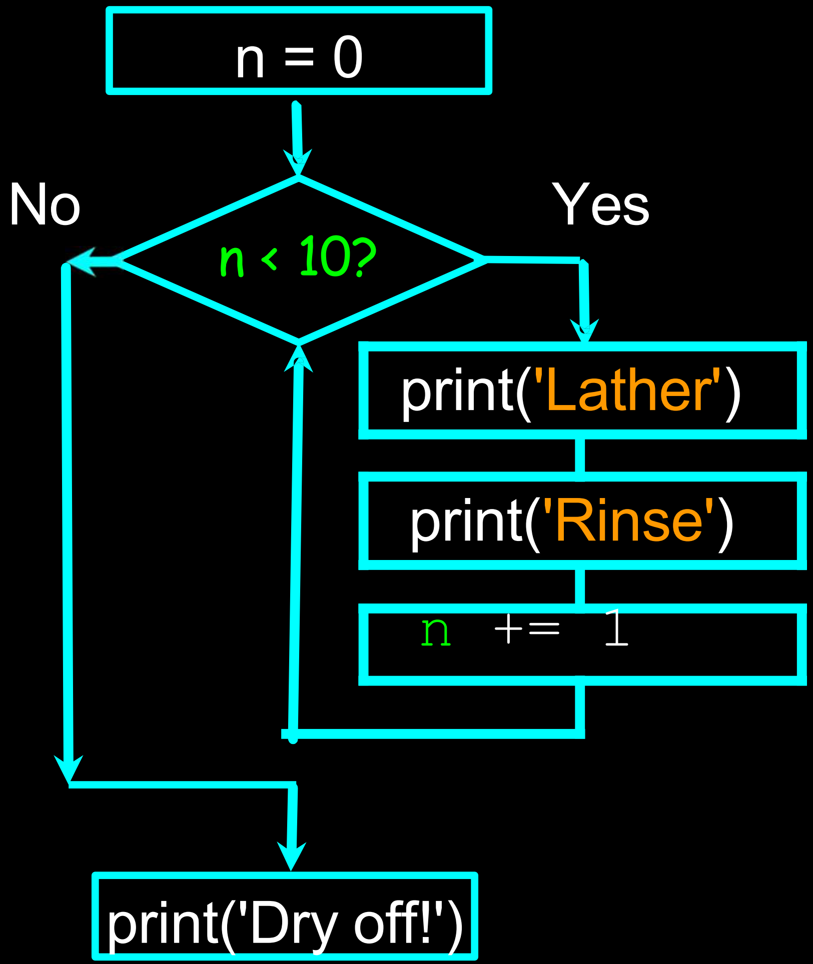
`while` *expression*`:`
    statements1
`else:`
    statements2

What is wrong with this loop?

An Infinite Loop



```
n = 5
while n > 0:
    print('Lather')
    print('Rinse')
print('Dry off!')
```



Another Loop

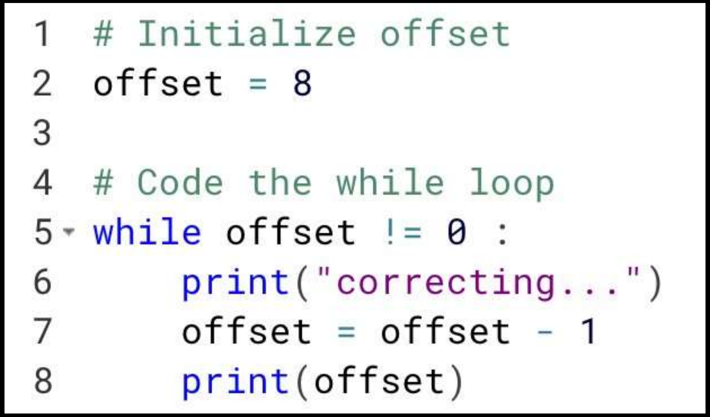
```
n = 0
while n < 10:
    print('Lather')
    print('Rinse')
    n += 1
print('Dry off!')
```

What is this loop doing?

in Python, most of the time **you will not** need a counter and a **`while`** loop, Since you'll use **`for`**.

Another Pattern for the `while` Loop

- In the below example, you will create the variable `offset` with an initial value of `8`. Then you will write a `while` loop that keeps running as long as the `offset` is not equal to `0`



```python
# Initialize offset
offset = 8

# Code the while loop
while offset != 0 :
    print("correcting...")
    offset = offset - 1
    print(offset)
```

```
correcting...
7
correcting...
6
correcting...
5
correcting...
4
correcting...
3
correcting...
2
correcting...
1
correcting...
0
```

https://www.datacamp.com/community/tutorials/python-while-loop

Your turn

What is the value of count after execution?

```
DNA_seq = ‘CTTACACACAAAAATAAT'

bp = ‘T'

count = 0
index = 0

while index < len(DNA_seq):
    if bp == DNA_seq[index]:
        count += 1
    index += 1

print('Our while count:', count)
```

```
# ------------------------------------------------------------------
# File name: while.py
#
# while EXPR:
#    statements
#
# while loop iterates the block of statements as long as EXPR
remains True.
#
# To illustrate the usage of while statement, the code below first
# computes the number of appearances of a nucleotide base in a
string
# using Python's str.count() method. Then it computes the same
number
# using a while statement, hoping to get the same answer.
#
# Version: 2.1
# Authors: H. Kocak and B. Koc
#         University of Miami and Stetson University
# References:
#
https://docs.python.org/3/reference/compound_stmts.html#the-
while-statement
# https://docs.python.org/3/library/stdtypes.html#string-methods
# https://www.ncbi.nlm.nih.gov/nuccore/KC545393.1?report=fasta
# ------------------------------------------------------------------
```

```
DNA_seq =
'CGGACACACAAAAAGAATGAAGGATTTTGAATCTTTATTGTGTGCGAG
TAACTACGAGGAAGATTAAAGA'
print('DNA sequence:', DNA_seq)

bp = 'T'
print('Base pair:', bp)

print('str.count():', DNA_seq.count(bp))

count = 0
index = 0

while index < len(DNA_seq):
   if bp == DNA_seq[index]:
      count += 1
   index += 1

print('Our while count:', count)
```

Loop Forever

A conditional’s loop expression can be as simple as a single true value, causing it to loop until an external event stops the program.

```
initialize values
while True:
   change values
   if test values:
      return
   use values
# repeat
return result
```

```
while True:
    line = input('> ')
    if line == 'done':
        print('You said', line)
    print(line)
print('Done!')
```

Breaking Out of a Loop

- `break` ends the current loop and jumps to the statement immediately following the loop
- It is like a loop test that can happen anywhere in the body of the loop
- But better to break early if you can

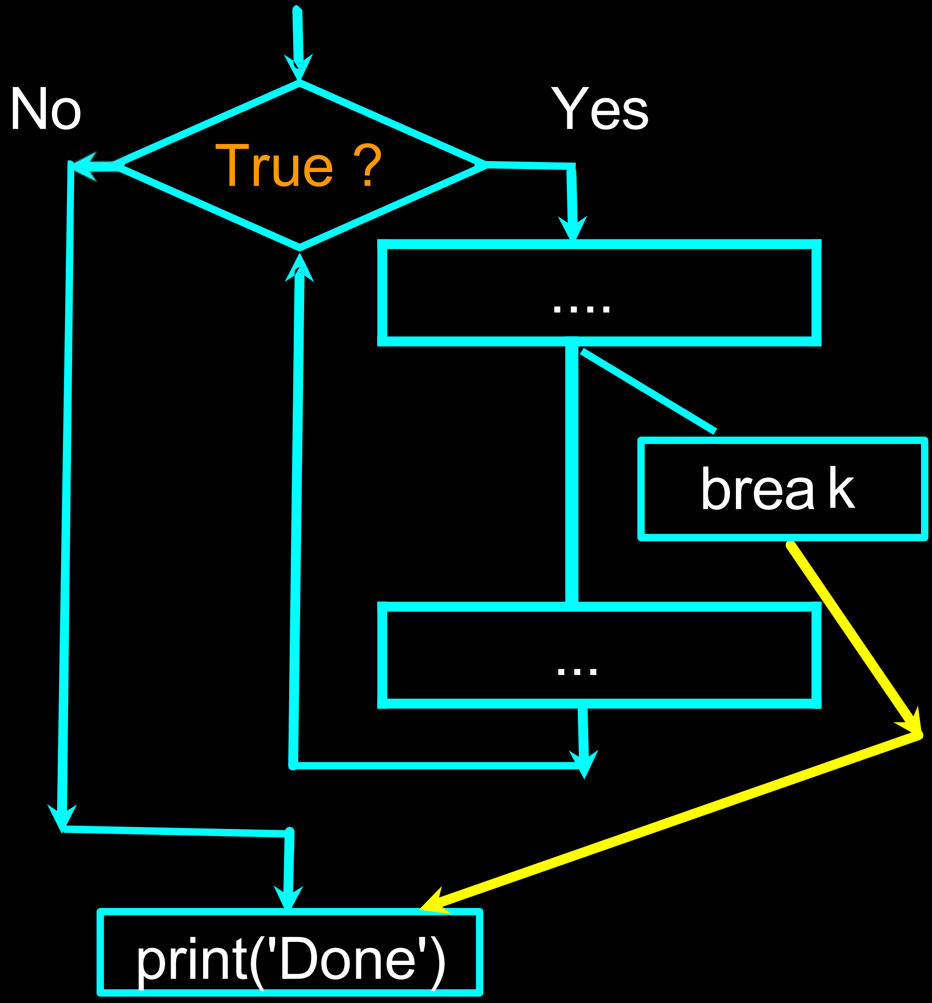
`while True:` and `while 1:`

```
while True:
    line = input('> ')
    if line == 'done':
        break
    print(line)
print('Done!')
```

```
> hello there
hello there
> finished
finished
> done
Done!
```

The Flow

```
while True:
    line = input('> ')
    if line == 'done':
        break
    print(line)
print('Done!')
```



continue

Last week we saw how `continue` ends the current iteration and jumps to the top of the loop and starts the next iteration

```
while True:
    line = input('> ')
    if line[0] == '#':
        continue
    elif line == 'done':
        break
    print(line)
print('Done!')
```

```
> hello there
hello there
> # don't print this
> print this!
print this!
> done
Done!
```

literation

Iteration

Python's for statement expresses iteration:

```
for item in collection:
   do something with item
```

Why iteration?

Imagine we wanted to take our list of apes:

```
apes = ["Homo sapiens", "Pan troglodytes", "Gorilla gorilla"]
```

and print out each element on a separate line, like this:

```
Homo sapiens is an ape
Pan troglodytes is an ape
Gorilla gorilla is an ape
```

One way to do it would be to just print each element separately:

```
print(apes[0] + " is an ape")
print(apes[1] + " is an ape")
print(apes[2] + " is an ape")
```

Writing a loop:

```
for ape in apes:
    print(ape + " is an ape")
```

Your turn:

Looping Through a List

What is the output?

```
print('Before')
for num in [9, 41, 12, 3, 74, 15]:
     print(num)
print('After')
```

Iteration Example

```python
apes = ["Homo sapiens", "Pan troglodytes", "Gorilla gorilla"]
for ape in apes:
    name_length = len(ape)
    first_letter = ape[0]
    print(ape + " is an ape. Its name starts with " + first_letter)
    print("Its name has " + str(name_length) + " letters")
```

Output:

```
Homo sapiens is an ape. Its name starts with H
Its name has 12 letters
Pan troglodytes is an ape. Its name starts with P
Its name has 15 letters
Gorilla is an ape. Its name starts with G
Its name has 15 letters
```

Your turn

```python
name = "martin"
for character in name:
    print("one character is \t" + character)
```

Option1:

```
one character is m
one character is a
one character is r
one character is t
one character is i
one character is n
```

Option2:

```
one character is 	m
one character is 	a
one character is 	r
one character is 	t
one character is 	i
one character is 	n
```

Iterating with `range` function

```python
for number in range(6):
    print(number)
```

```
0
1
2
3
4
5
```

```python
for number in range(3, 8):
    print(number)
```

```
3
4
5
6
7
```

```python
for number in range(2, 14, 4):
    print(number)
```

```
2
6
10
```

Iterating with an index

2. Loop over the range of indices
   - `for i in range(len(name)):`
   - Inside the loop, `name[i]` gives the character at that index
   - But we learned last week to avoid using `range(len(name))`

The Pythonic way would be to use the built-in function **`enumerate`**

3. Use **`enumerate`** to get both character and index at the same time
   - `for pos, char in enumerate(name):   # important have two vars here`
   - Each iteration, `pos` will be the index
   - … and `char` will be the character at that position

A **Counter (Index)** and **Value** in Python

Remember: Python eases the programmers' task by providing a built-in function `enumerate()` for this task. `enumerate()` method adds a counter to an iterable and returns it in a form of an `enumerate` object

```
for counter, num in enumerate([9, 41, 12, 3, 74,  15]):
   print(counter, num)
print('After', counter)
```

```
$ python enumerate_.py
0 9
1 41
2 12
3 3
4 74
5 15
After 5
```

Using `enumerate` last number of `counter` is N-1 the size of the iterable by default (`start=0`)

Easily change the start count/index with help of `enumerate(sequence, start=1)`

Use Two Variables with `enumerate`

```
>>> for pos, char in enumerate(string):
...     print(pos, type(pos), char, type(char))
...
1 <class 'int'> a <class 'str'>
2 <class 'int'> b <class 'str'>
3 <class 'int'> c <class 'str'>
4 <class 'int'> d <class 'str'>
```

Why both `pos` and `char` here?

```
>>> for char in enumerate(string):
... print(char, type(char))
...
(0, 'a') <class 'tuple'>
(1, 'b') <class 'tuple'>
(2, 'c') <class 'tuple'>
(3, 'd') <class 'tuple'>
```

In the solution above Python unpacks the tuple into two variables (`pos` and `char`), if you don't do this you'll have to access via the tuple, e.g. `tuple[0]` or `tuple[1]`. Convention is to unpack

Compare the Two Options

- Both produce the same output
  - But using a `for` loop (definite loop) is much more elegant
- The iteration variable is completely taken care of by the `for` loop

```
index = 0
while index < len(DNA_seq):
    letter = DNA_seq[index]
    print(letter)
    index = index + 1
```

Much simpler!

```
fruit = 'CGGAC'
for letter in
    DNA_seq:
    print(letter)
```

```
C
G
G
A
C
```

if you need an **`index` like variable**, use `enumerate`, as we'll see on an upcoming slide

Looping and Counting

This is a simple loop that loops through each letter in a string and counts the number of times the loop encounters the 'a' character

Of course, we could just use `str.count()` to get the # `a`'s in the sequence

```
>>> a = 'aaaba'
>>> a.count('a')
4
```

but think of the pattern here, more than counting the characters in the string. We'll learn loop patterns next week

```
word = ‘CGGAC’
count = 0
for letter in word:
    if letter == 'C':
        count += 1

print(count)
```

Loop Idioms:

Note: Even though those examples are simple, the patterns apply to all kinds of loops

Making “smart” loops

The trick is “knowing” something about the whole loop when you are stuck writing code **that only sees one entry at a time**

It's very easy at times, and quite challenging as you progress as a programmer

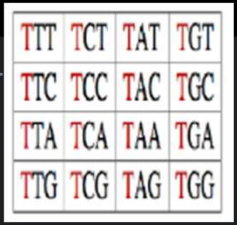
```
Set some variables to initial values

for thing in data:
    Look for something or do something to each entry separately, updating a variable

Look at the variables
```

```
# ------------------------------------------------------------
# File name: for.py
#
# for item in items:
#     statements

# Python’s for statement iterates over the items of any
sequence
# (a list or a string), in the order that they appear in the
sequence.
#
# Code below prints out all codons starting with T.
#
# Version: 2.1
# Authors: H. Kocak and B. Koc
#          University of Miami and Stetson University
# References:
# https://docs.python.org/3/tutorial/controlflow.html#for-
statements
#
https://docs.python.org/3/reference/compound_stmts.html#f
or
# https://en.wikipedia.org/wiki/DNA_codon_table
```



TTT	TCT	TAT	TGT
TTC	TCC	TAC	TGC
TTA	TCA	TAA	TGA
TTG	TCG	TAG	TGG

```
# ------------------------------------------------------------

# Save nucleotide bases in a list
bases = ['T', 'C', 'A', 'G']

# As a warmup, print the list of bases
for base in bases:
   print(base)

print('Codons starting with T:')

for second_base in bases:
   print('Codons starting with T'+second_base)
   for third_base in bases:
      print('T'+second_base+third_base)
```

When to use `for` or `while`

- Use a **`for`** loop if you know, ***before you start looping***, the maximum # of times that you'll need to execute the body
  - Like traversing a list of elements, you know that the maximum number of loop iterations you can possibly need is "***all the elements in the list***"
  - Or if you need to print the "12 times table", i.e. we know right away how many times the loop will need to run
  - So any problem like "iterate this weather model for 1000 cycles", or "search this list of kmers", "find all prime numbers up to 10000", All suggest that a `for` should be used

- By contrast, if you are required to repeat some computation ***until some condition is met***, and ***you cannot calculate in advance when*** (or if) this will happen, you'll need a **`while`** loop
- If you need a process to run like a daemon (https://en.wikipedia.org/wiki/Daemon_(computing))

- **`for`** = ***definite iteration*** — we know ahead of time some definite bounds for what is needed
- **`while`** = ***indefinite iteration*** — we're not sure how many iterations we'll need — we cannot even establish an upper bound!

Jeffrey Elkner