CSE 101

Computer Science Principle

Lecture 04: Iterations, Lists & Algorithm design

March. 2023
Prof. Francois Rameau
francois.rameau@sunykorea.ac.kr

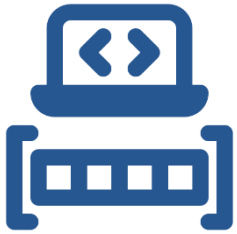
SUNY Korea
The State University of New York

Attendance time!

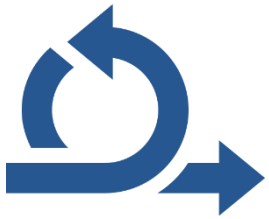
Scan the QR code and select your name in the list

What are we studying today?

This week we will address a few more **fundamental concepts** in programming!



Lists



Iterations

Debugging

Welcome to the loop, where we'll be diving into the world of code repetition and exploring the power of loops and iterations.

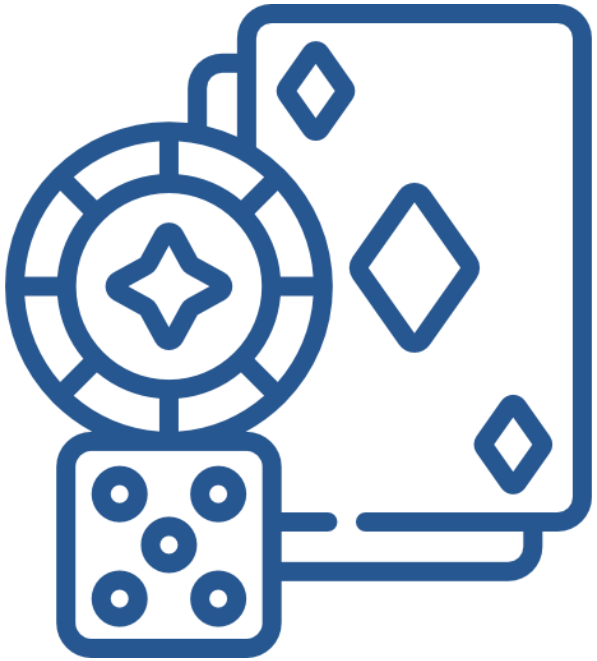
Loops and iterations

Let's play a game

Now that you have some basis in programming let's program a simple game together

Let's build a gambling machine:

1. A random integer between 0 to 10 is generated
2. The player has to guess the number and can win the grand prize



Let's play a game

The code is simple!

```
import random

# Generate a random integer between 1 and 10 (inclusive)
number_to_guess = random.randint(1, 10)

# input from the user and check if it is a win
user_number = int(input("enter a number between 0 to 10 : "))
if (number_to_guess!=user_number):
    print("you loose")
else:
    print("you win")
```

Let's play a game

It is now time for an impressive live demo!

We can identify two problems with this games:

1. The player cannot replay multiple times in a row, he has only one chance
2. The random number is changing every time we rerun the code

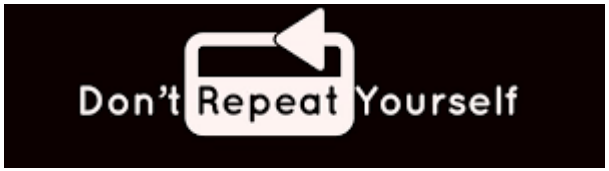
Let's play a game

No problem! Let's fix it, to give the player more chance to play

```python
import random

# Generate a random integer between 1 and 10 (inclusive)
number_to_guess = random.randint(1, 10)

# First trial
print("### trial 1 ###")
user_number = int(input("enter a number between 0 to 10 : "))
if (number_to_guess!=user_number):
    print("you loose")
else:
    print("you win")

# Second trial
print("### trial 2 ###")
user_number = int(input("enter a number between 0 to 10 : "))
if (number_to_guess!=user_number):
    print("you loose")
else:
    print("you win")


# Third trial
print("### trial 3 ###")
user_number = int(input("enter a number between 0 to 10 : "))
if (number_to_guess!=user_number):
    print("you loose")
else:
    print("you win")
```

Now the player has three chances to win!

Is it a correct way to address this problem? What if I now want to give him 15 chances?



Iterations

Thankfully programming is all about **iterations**

! Terminology alert !

Iteration: In programming, iteration is the act of repeating a set of instructions multiple times. It is achieved using looping constructs, to execute the same block of code multiple times with different inputs or parameters.

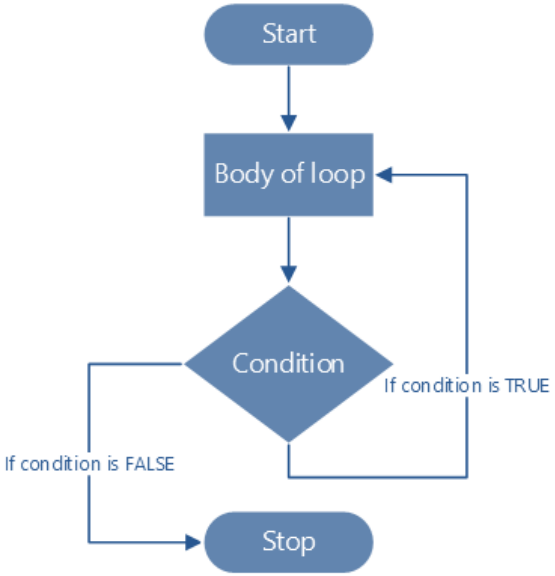
Let's introduce the mighty `while` and `for` loops


- While loop -

While loop

! Terminology alert !

While loop: A while loop is a programming structure that allows a block of code to be repeated until a certain condition is no longer true.



Example

```
count = 0
while count < 10:
    print("Count is:", count)
    count += 1
```

What is this code doing exactly?

Let's run it!

While loop: Control flow

Pseudo Code

```
while<condition>
    # Run the block of code
    <expression>
    <expression>
    ...
    <expression>
```

- `<condition>` evaluates a Boolean
- If `<condition>` is `True`, do all the steps inside the while block
- Check `<condition>` again
- Repeat until `<condition>` is `False`

While loop: another example

Let's say you now want to create a system collecting data, the user has to enter a few personal information. But you want your system to be able to detect unintended inputs

```
correct_name = False
while (correct_name == False):
    #input name
    name = input("please, enter your name : ")
    if (name==""):
        print("you did not entered your name properly, do it again")
    else:
        print("Hello " + name)
        correct_name = True
```

What will happen if we remove the last line? Is it a wanted behavior?

While loop: infinite loop

I designed a tiny video game. The character can move on any empty space by pressing directional arrows

The character can move indefinitely in this maze. The pseudo code looks like this

```
while True:
        1. Detect pressing directional key
        2. Move Left/right/down/up if possible
        3. display the maze with the robot new position
```

While loop: Nested loops

```
''' --- Nested while loops example ---
Print once the word "hello“
and 4 time the word "world".
This operation will run 5 times
'''

i = 0
while (i < 5):
    print("--- iteration {}---".format(i+1))
    print("Hello")
    i = i + 1
    j = 0
    while (j < 4):
        print("World")
        j+=1
```

```
--- iteration 1---
Hello
World
World
World
World
--- iteration 2---
Hello
World
World
World
World
--- iteration 3---
Hello
World
World
World
World
--- iteration 4---
Hello
World
World
World
World
--- iteration 5---
Hello
World
World
World
World
```

Can you guess the output??

While loop: Nested loops

We can also count backward!

```
i = 5
while (i > 0):
    print("--- iteration {}---".fo
rmat(i))
    print("Hello")
    i = i - 1
    j = 4
    while (j > 0):
        print("World")
        j-=1
```

```
--- iteration 5---
Hello
World
World
World
World
--- iteration 4---
Hello
World
World
World
World
--- iteration 3---
Hello
World
World
World
World
--- iteration 2---
Hello
World
World
World
World
--- iteration 1---
Hello
World
World
World
World
```

Let's try a debugger on it? To better understand its behavior

Break the while loop!

Is the while loop will iterate forever until its condition is respected?

Not necessarily, let's break the loop

```
condition = True
while (condition):
    print("I am stuck here forever !!")
```

This code will run forever

```
condition = True
iteration_number = 0
while (condition):
    print("I am stuck here forever !!")
    iteration_number += 1
    if (iteration_number>6):
        break
print("I am out!")
```

```
I am stuck here forever !!
I am stuck here forever !!
I am stuck here forever !!
I am stuck here forever !!
I am stuck here forever !!
I am stuck here forever !!
I am stuck here forever !!
I am out!
```

Break terminates the loop completely!

Terminate the current iteration

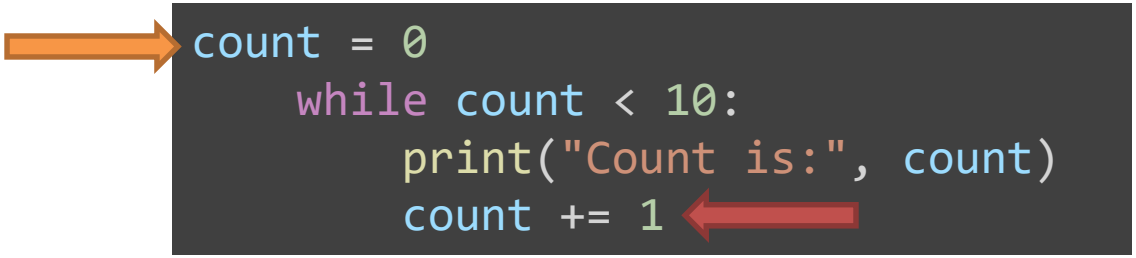
`break` exits the entire loop, if you just want to terminate the current iteration early you can use **`continue`**

```
iteration_number = 0
while (iteration_number<10):
    iteration_number += 1
    #is the iteration number odd?
    if (iteration_number % 2) == 0:
        continue
    print("Here is a odd number " + str(iteration_number))
```

```
Here is a odd number 1
Here is a odd number 3
Here is a odd number 5
Here is a odd number 7
Here is a odd number 9
```

While loop: limitations

We have seen that while loop can be used for counting



```
count = 0
    while count < 10:
        print("Count is:", count)
        count += 1
```

But if you do that, you need to take care of two things:

1. The initialization of the counter
2. The increment (if you forget to update you increment, we are stuck in an infinite loop)

FOR loop are much better for counting things! (and more)

List

Before we continue talking about the FOR loops, let's first introduce the concept of **lists**

- Debugging time -

Inspired by LinkedIn class on Debugging

Debugging

As your code are getting more complex (especially with loops), it is always good to know debugging tools!

! Terminology alert !

A **debugger/trace software** allows programmers to step through their code, inspect variables and memory, set breakpoints, and analyze program behavior. Debuggers are commonly used in software development to identify and fix bugs in programs.

Debugging

The debugging philosophy is somewhat similar to your thought process when repairing a computer

- Check if the computer is plugged
- Check if the screen is functioning
- Check if the fans are turning
- Check if the Hard drive is properly plugged
- Etc…

This is a logical procedure that consist in: 1. Identifying the problem, 2. Gather information, 3. Check for obvious issues, 4. Use diagnostic tools, 5. Test components individually, 6. Fix the issue

Debugging

Just as a computer, a program does not always work as expected or may not work at all!

But each problems are unique

The process of identifying and fixing bugs is called debugging

Main types of bug

Syntax Error

Run-time Error

Logic error

Syntax Error

Syntax Error: Code does not match the rules of the language

Try to spot the error

Missing ":"

```
rain = True
if rain==True
    print("Bring an umbrella")
```

```
File "./weather_conditional.py", line 2
 if rain==True
             ^
SyntaxError: invalid syntax
```

Now do you see it?

Run-time Error

Run-time Error: Your code complies with all the rules of the language but still cannot execute

Try to spot the error

```
rain = True
isUmbrellaNeeded(rain)
```

```
Traceback (most recent call last):
  File ".\nested_while_loops.py", line 40, in <module>
    isUmbrellaNeeded(rain)
NameError: name 'isUmbrellaNeeded' is not defined
```

```
def isUmbrellaNeeded(rain):
    if rain==True:
        print("Bring an umbrella")
    else:
        print("You do not need an umbrella")

rain = True
isUmbrellaNeeded(rain)
```

Logic Error

Logic Error: Your code is valid but it does not do what you wanted to achieve

Try to spot the error

Goal: You want to count from 10 to 0

```
i = 10
while i>0:
    print(i)
    i+=1
```

OH! No … the loop is running to infinity

```
i = 10
while i>0:
    print(i)
    i-=1
```

Multiple bugs

It is very likely that your code will contain multiple bugs!

```
def isUmbrellaNeeded(rain)
    if rain==True:
        print("Bring an umbrella"
    else:
        print("You do not need an umbrella")

rain = True
isUmbrellaNeeded(rain)
```

You will be able solve the error one after another in order of appearance

```
File "test_fct.py", line 13
   def isUmbrellaNeeded(rain)
                        ^
SyntaxError: invalid syntax
```

1

```
File "test_fct.py", line 16
   else:
   ^
SyntaxError: invalid syntax
```

2

Debugging

Logic error are often the most difficult to debug because the error message cannot help you

No worries

Debugging is part of programming and every single language have tools to help you to fix the bugs! You just need to practice and learn the tools we can use!

First of all modern IDE contain various tools to avoid you to make error in a first place!

Highlight

Autocomplete

Linter

Trace

Tracking the behavior of this code can be particularly complex

You can also use some logging that activates only during debug stage

```
def function_1(input_val):
    val_1 = input_val**2 +4
    val_2 = function_3(val_1)
    return val_2
def function_2(input_val):
    return input_val**0.5
def function_3(input_val):
    val_1 = input_val/25
    val_2 = function_2(val_1)
    return val_2

a = 5
result = function_1(a)
print(result)
```

One way for debugging is to use many prints but ….

```
def function_1(input_val):
    print("Enter function_1")
    val_1 = input_val**2 +4
    val_2 = function_3(val_1)
    return val_2
def function_2(input_val):
    print("Enter function_2")
    return input_val**0.5
def function_3(input_val):
    print("Enter function_3")
    val_1 = input_val/25
    val_2 = function_2(val_1)
    return val_2

a = 5
result = function_1(a)
print(result)
```

Trace

Recent IDE contains trace options! How to use it?

1. Set a breakpoint by clicking on the line number

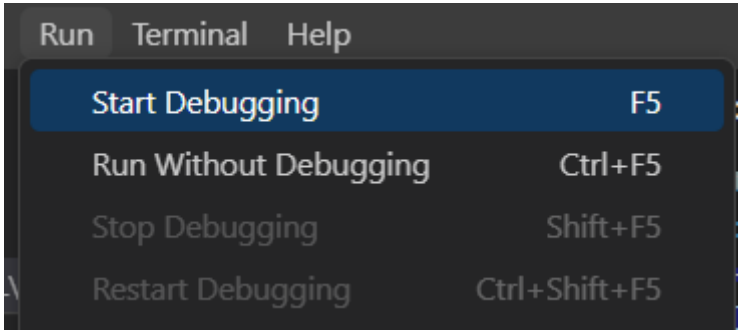
C: > Users > DELL > Documents > Code > test_function >

```
1   def function_1(input_val):
2       val_1 = input_val**2 +4
3       val_2 = function_3(val_1)
4       return val_2
5   def function_2(input_val):
6       return input_val**0.5
7   def function_3(input_val):
8       val_1 = input_val/25
9       val_2 = function_2(val_1)
10      return val_2
11
12  a = 5
13  result = function_1(a)
14  print(result)
```

In debugger tools, a "**breakpoint**" is a marker that you can set in your code to pause the execution of the program at a specific line or statement. When the program reaches the breakpoint, it stops executing and gives you the opportunity to inspect the program's state and variables, and step through the code line by line to understand how the program is running.

Trace

2. Then run the code in debugging mode !

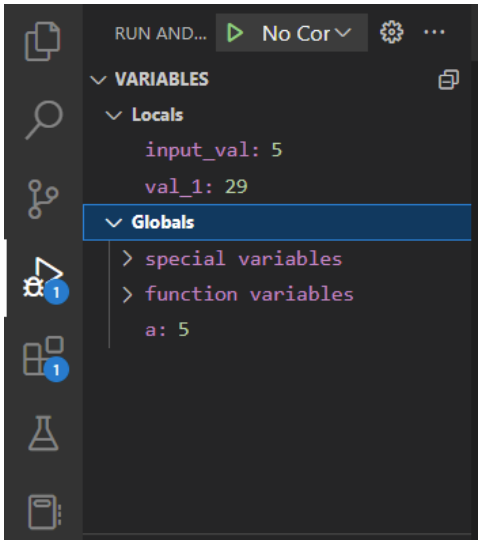


3. The code will stop at your breakpoint

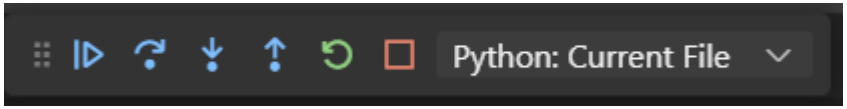
```
1   def function_1(input_val):
2       val_1 = input_val**2 +4
3       val_2 = function_3(val_1)
4       return val_2
5   def function_2(input_val):
6       return input_val**0.5
7   def function_3(input_val):
8       val_1 = input_val/25
9       val_2 = function_2(val_1)
10      return val_2
11
12  a = 5
13  result = function_1(a)
14  print(result)
```

Trace

Now on the left side of the screen you can see an overview of all the local and global variables current in memory!



You can also execute the code step by step thanks to the control bar (Let's demo it!)



Trace

Thanks to the python pdb you can debug without IDE!

1. Import the pdb module

2. Set you breakpoint by adding the line pdb.set_trace()

```
import pdb

def function_1(input_val):
    val_1 = input_val**2 +4
    pdb.set_trace()
    val_2 = function_3(val_1)
    return val_2
def function_2(input_val):
    return input_val**0.5
def function_3(input_val):
    val_1 = input_val/25
    val_2 = function_2(val_1)
    return val_2

a = 5
result = function_1(a)
print(result)
```

Then run the code as usual, and it will stop at the breakpoint (let's test together)

Trace

A few pdb debugging functions to navigate in the code

s(tep)
Execute the current line, stop at the first possible occasion (either in a function that is called or on the next line in the current function).

n(ext)
Continue execution until the next line in the current function is reached or it returns. (The difference between `next` and `step` is that `step` stops inside a called function, while `next` executes called functions at (nearly) full speed, only stopping at the next line in the current function.)

unt(il) `[lineno]`
Without argument, continue execution until the line with a number greater than the current one is reached.

With a line number, continue execution until a line with a number greater or equal to that is reached. In both cases, also stop when the current frame returns.

Changed in version 3.2: Allow giving an explicit line number.

r(eturn)
Continue execution until the current function returns.

c(ont(inue))
Continue execution, only stop when a breakpoint is encountered.

j(ump) `lineno`
Set the next line that will be executed. Only available in the bottom-most frame. This lets you jump back and execute code again, or jump forward to skip code that you don't want to run.

It should be noted that not all jumps are allowed – for instance it is not possible to jump into the middle of a `for` loop or out of a `finally` clause.

l(ist) `[first[, last]]`
List source code for the current file. Without arguments, list 11 lines around the current line or continue the previous listing. With `.` as argument, list 11 lines around the current line. With one argument, list 11 lines around at that line. With two arguments, list the given range; if the second argument is less than the first, it is interpreted as a count.

The current line in the current frame is indicated by `->`. If an exception is being debugged, the line where the exception was originally raised or propagated is indicated by `>>`, if it differs from the current line.

New in version 3.2: The `>>` marker.

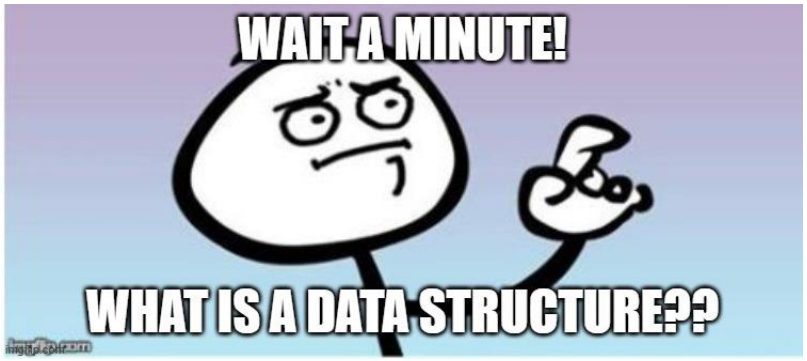
ll | `longlist`
List all source code for the current function or frame. Interesting lines are marked as for `list`.

- Lists -

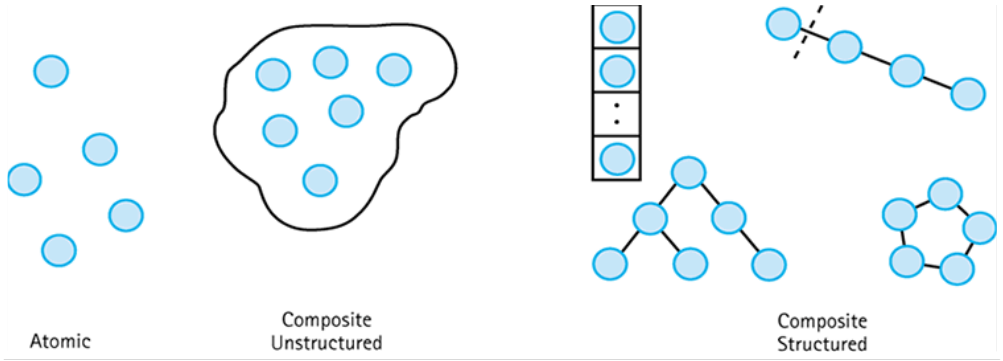
Inspired by:
Socratica – YouTube

Data structure

We will now see our first data structure!

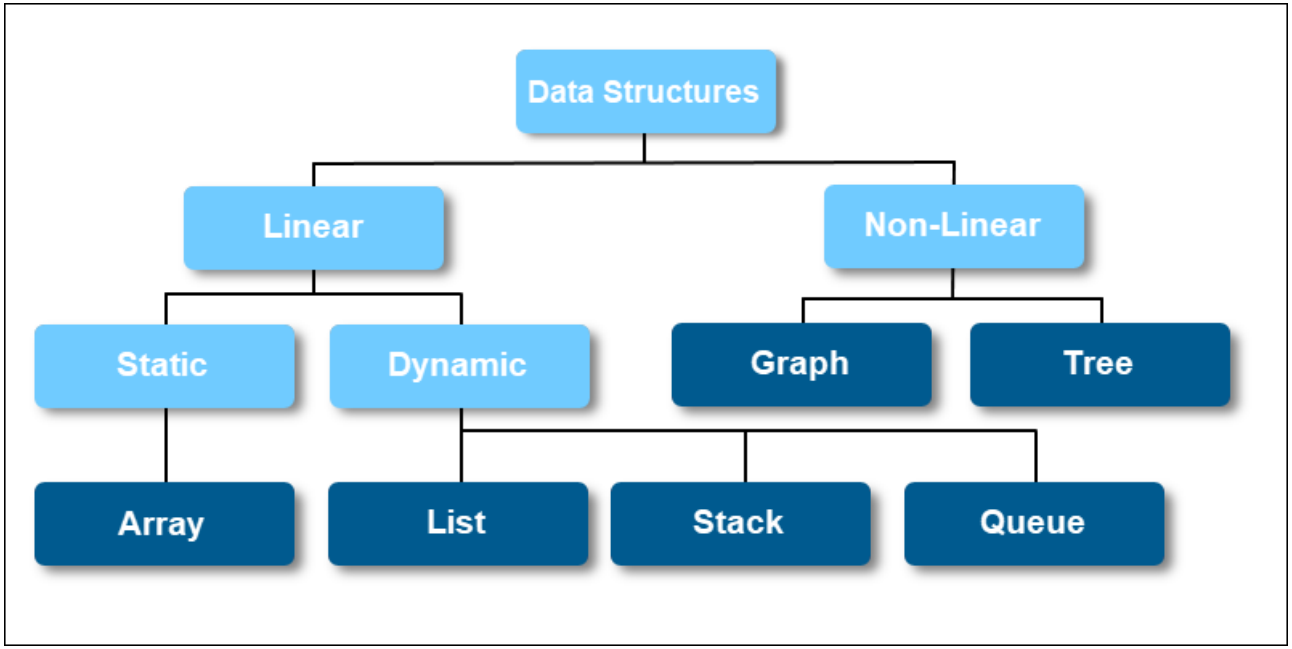


Data structure = A way of organizing data



Data structure

Plenty of different data structure exist, today we will study the **lists**

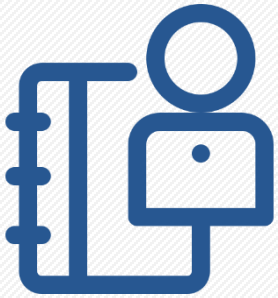


! Terminology alert !

A **data structure** is a way of organizing and storing data in a computer program so that it can be accessed and manipulated efficiently. It provides a particular way of representing and manipulating data in a structured and organized manner.

What is a list?

In our everyday life, it is common to use lists to organize data

Grocery list	To-do list	Student list
		
Bank statement	**Team management**	**inventory**
		

What is a list?

Why do we use lists so much?

Because lists make it easy to remember things and organize information in a given order

Similarly, in programming, this is one of the most common data structures which simplify the storage and manipulation of **ordered data**

! Terminology alert !

A **list** is a common data structure that allows you to store an ordered collection of items. Each item in the list is identified by its index, which is a number that starts at 0 for the first item, 1 for the second item, and so on.

How to create a list?

As you understood, a list allows storing multiple items in a single variable

Create a list!

```
# We can create an empty list using the constructor
My_list = list()
# More commonly, we will use []
my_list = []
# You can also initialize your list with values/items
my_list = [89,4,23,76]
```

Access an item in the list

The function len() return the length of a list

value	89	4	23	76
index	0	1	2	3

my_list

```
len(my_list)
```


4

Important note

- If a list has n item, the locations in the list are numbered from 0 to n-1 not 1 through n)
- The notation `a[i]` stands for "the item at location `i` in list a
- In programming, use the word **index** to refer to the numerical position of an element in a list

You can access every single item in the list via its index

value	89	4	23	76
index	0	1	2	3

my_list

```
print(my_list[1])
```


4

Guess the value returned by index 3!

```
print(my_list[4])
```


IndexError: list index out of range

Access an item in the list

You can also access elements from the end of the list by using negative numbers (because python wrap back around the end of the list)

value	89	4	23	76
index	0	1	2	3
neg index	-4	-3	-2	-1

`my_list`

```
>>> my_list[-1]
76
>>> my_list[-2]
23
>>> my_list[-3]
4
>>> my_list[-4]
89
>>> my_list[-5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Access an item in the list

You can also access all items in a range of index (it I called **slicing**)

value	89	4	23	76	22	85	4
index	0	1	2	3	4	5	6

If you want to access to the list of item stored from index 2 to 5, you can use:

```
sub_list = my_list[2:5]
```

➡ [23, 76, 22]

If you do not specify the beginning or the end, it will go all the way through

```
sub_list = my_list[2:]
```

➡ [23, 76, 22, 85, 4]

```
sub_list = my_list[:5]
```

➡ [89, 4, 23, 76, 22]

Add item in a list

Imagine a list of the six first prime number

```
#initialize a list of the six first prime number
prime = [2, 3, 5, 7, 11, 13]
```

What if I want to add a seventh one?

```
# add one more prime number to the list
prime.append(17)
```

And another one?

```
prime.append(19)
```

How does it looks like now?

```
print(prime)
```

➡ [2, 3, 5, 7, 11, 13, 17, 19]

Add item in a list

Instead of using the method `append()` to add an element, you can use the operator **+=**

```
prime += [23]
```

```
print(prime)
```

➡ [2, 3, 5, 7, 11, 13, 17, 19, 23]

Or multiple elements at once

```
prime += [29, 31]
```

➡ [2, 3, 5, 7, 11, 13, 17, 19, 23]

What data can I store in a list?

A list can store any sort of objects!

Any sort? Yes!

List of int

```
list_int = [55,-10, 23, 4]
print(list_int)
```

⬇

[55, -10, 23, 4]

List of float

```
list_float = [0.22, 10.26, 5.1, 2]
print(list_float)
```

⬇

[0.22, 10.26, 5.1, 2]

List of Boolean

```
list_bool = [True, True, True, False]
print(list_bool)
```

⬇

[True, True, True, False]

List of string

```
fruits =['banana', 'apple', 'grape’]
print(fruits)
```

⬇

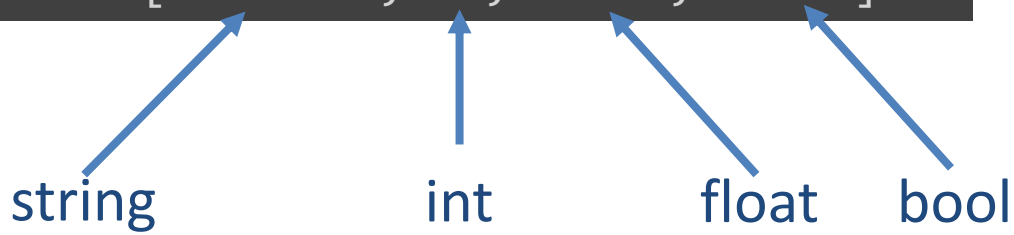
['banana', 'apple', 'grape']

What data can I store in a list?

You can even mix different types of data

Image you want to store the name, age, weight and marital status of a person

```
person_data = ['Arthur', 46, 70.54, False]
```



You can even create a list of lists!

Now imagine you want to store the data of multiple persons

```
persons_data = [['Chaoning', 46, 70.54, False], ['Mary', 23, 50.33, True], \
                ['George', 23, 90.00, True], ['Mohamed', 55, 75.93, True]]
```

Access data in a nested list

What is concretely a list of list?

```
nexted_list = [[1, 2, 3][4, 5, 6]]
```

And now if I rewrite it as follow?

```
nexted_list = [[1, 2, 3],
               [4, 5, 6]]
```

It can be seen as a **matrix/array**

How can we access the individual values inside?

```
matrix = [[1, 2, 3],
          [4, 5, 6]]
first_row = matrix[0]
second_row = matrix[1]
one_element = matrix[1][1]
print(first_row)
print(second_row)
print(one_element)
```

Concatenate lists of different types

As we have seen before can concatenate lists using the + sign

It is also working for lists containing different type of data

```
numbers = [1, 2, 3]
text = ['one', 'two', 'three']
print(numbers + text)
```

➡ [1, 2, 3, 'one', 'two', 'three']

Order matters

```
numbers = [1, 2, 3]
text = ['one', 'two', 'three']
print(text + numbers)
```

➡ ['one', 'two', 'three', 1, 2, 3,]

Methods/Functions on lists

You can see all the methods and functions existing on list by typing `dir(your_list)`

```
>>> numbers = [1, 2, 3]
>>> dir(numbers)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__',
'__iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__',
'__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append',
'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

You can get information about any specific using `help`

```
>>> help(numbers.count)
Help on built-in function count:

count(value, /) method of builtins.list instance
    Return number of occurrences of value.
```

```python
print("--- 1. Declare a list ---")
numbers = [1, 2, 3, 4, 5, 1, 2, 3]
print(numbers)
# 1. copy() -- > Copy the list
print("--- 1. Test copy ---")
numbers_copy = numbers.copy()
numbers_copy[3] = 50
print(numbers_copy)
# 2. count(value) --> Returns the number of elements with the specified value
print("--- 2. Count occurences ---")
nb_of_ones = numbers.count(1)
print(nb_of_ones)
# 3. extend(list) --> Add the elements of a list (or any iterable), to the end of the current
list
print("--- 3. Extend/Concatenate ---")
numbers.extend(numbers_copy)
print(numbers)
# 4. index(value) --> Returns the index of the first element with the specified value
print("--- 4. search index of a value ---")
index_fifty = numbers.index(50)
print(index_fifty)
# 5. insert(index, value) --> Adds an element at the specified position
print("--- 5. insert a value at a given index ---")
numbers.insert(3, 4.5)
print(numbers)
# 6. pop(index) --> Removes the element at the specified position
print("--- 6. remove a value at a given index ---")
numbers.pop(5)
print(numbers)
# 7. remove(value) --> Removes the first item with the given value
print("--- 7. remove first occurence of a specific value---")
numbers.remove(3)
print(numbers)
# 8. reserve() --> Reverses the order of the list
print("--- 8. reverse order---")
numbers.reverse()
print(numbers)
# 9. sort() --> sort the list
print("--- 9. sort---")
numbers.sort()
print(numbers)
# 10. clear() --> erase everything in the list
print("--- 10. clear---")
numbers.clear()
print(numbers)
```

Output

```
--- 1. Declare a list ---
[1, 2, 3, 4, 5, 1, 2, 3]
--- 1. Test copy ---
[1, 2, 3, 50, 5, 1, 2, 3]
--- 2. Count occurrences ---
2
--- 3. Extend/Concatenate ---
[1, 2, 3, 4, 5, 1, 2, 3, 1, 2, 3, 50, 5, 1, 2, 3]
--- 4. search index of a value ---
11
--- 5. insert a value at a given index ---
[1, 2, 3, 4.5, 4, 5, 1, 2, 3, 1, 2, 3, 50, 5, 1, 2, 3]
--- 6. remove a value at a given index ---
[1, 2, 3, 4.5, 4, 1, 2, 3, 1, 2, 3, 50, 5, 1, 2, 3]
--- 7. remove first occurrence of a specific value---
[1, 2, 4.5, 4, 1, 2, 3, 1, 2, 3, 50, 5, 1, 2, 3]
--- 8. reverse order---
[3, 2, 1, 5, 50, 3, 2, 1, 3, 2, 1, 4, 4.5, 2, 1]
--- 9. sort---
[1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 4, 4.5, 5, 50]
--- 10. clear---
[]
```

A few useful functions

```
# Declare a list
print("--- 1. Declare a list ---")
numbers = [1, 2, 3, 4, 5, 1, 2, 3]
print(numbers)
# Min
print("--- 1. find minimum value in the list ---")
min_value = min(numbers)
print(min_value)
# Max
print("--- 2. find max value in the list ---")
max_value = max(numbers)
print(max_value)
# Sum
print("--- 3. find the sum of all value ---")
sum_value = sum(numbers)
print(sum_value)
```

```
--- 1. Declare a list ---
[1, 2, 3, 4, 5, 1, 2, 3]
--- 1. find minimum value in the lis
1
--- 2. find max value in the list ---
5
--- 3. find the sum of all value ---
21
```

Copying a list

Be very careful when you copy a list!!

```
list_1 = [1, 2, 3]
list_2 = list_1
list_2[1]= 500
print(list_1)
```

[1, 500, 3]

Using the operator "=" copy the addresses so modifying one list will affect the other

```
list_1 = [1, 2, 3]
list_2 = list_1.copy()
list_2[1]= 500
print(list_1)
```

[1, 2, 3]

Tuples

Python proposes other data structures to deal with ordered data

(TUPLES)

! Terminology alert !

A **tuple** is an ordered collection of elements, similar to a list, but with the key difference that tuples are immutable - once created, the elements of a tuple cannot be modified

Tuples support much less methods than list!

So why do we use them??

Speed

Memory

Tuples

How to declare a tuple?

Empty tuple	`my_tuple = ()`
Sequence of numbers	`scores = (90, 60, 88, 95, 23)`
Sequence of strings	`letters = ("a", "b", "c")`
Sequence of lists	`matrix = ([1,2], [3,4])`

How do you access the elements inside?

Exactly like in a list!

`scores[1]`	60
`letters[2]`	'c'
`matrix[0][1]`	2

Tuples

What methods are implemented in tuples?

ONLY TWO OF THEM

`count()`

Returns the number of times a specified value occurs in a tuple

`index()`

Searches the tuple for a specified value and returns the position of where it was found

list has much more methods than tuples

But list requires more memory and is slower

Tuples

Let's try to modify an element in a tuple!

```
data = (1, 2, 3)
data[1] = 6
```

```
File ".\function_list.py", line 34, in <module>
    data[1] = 6
TypeError: 'tuple' object does not support item assignment
```

It cannot be changed!

It is immutable

Tuples

[ ] List

- Add data
- Change data
- Remove data

( ) Tuple

- Cannot be changed
- Immutable
- Made quickly

Tuples

Alternative construction of tuples

```
data_1 = 1,
data_2 = 1, 2, 3
data_3 = 4, 5, 6, 7
```

Note the peculiar construction for the single element tuple

Tuples – return multiple variables

We have already briefly encounter tuples in this class

Do you remember the multi-output functions?

```
def my_function ():
    a = 1
    b = 2
    return a, b

outputs = my_function()
print(type(outputs))
```

```
<class 'tuple'>
```

Thus I can access each variable individually as follow:

```
outputs[0]
outputs[1]
```

Tuples – return multiple variables

But wait, we can also retrieve the parameters of this function like this?

```
def my_function ():
    a = 1
    b = 2
    return a, b

a, b = my_function()
```

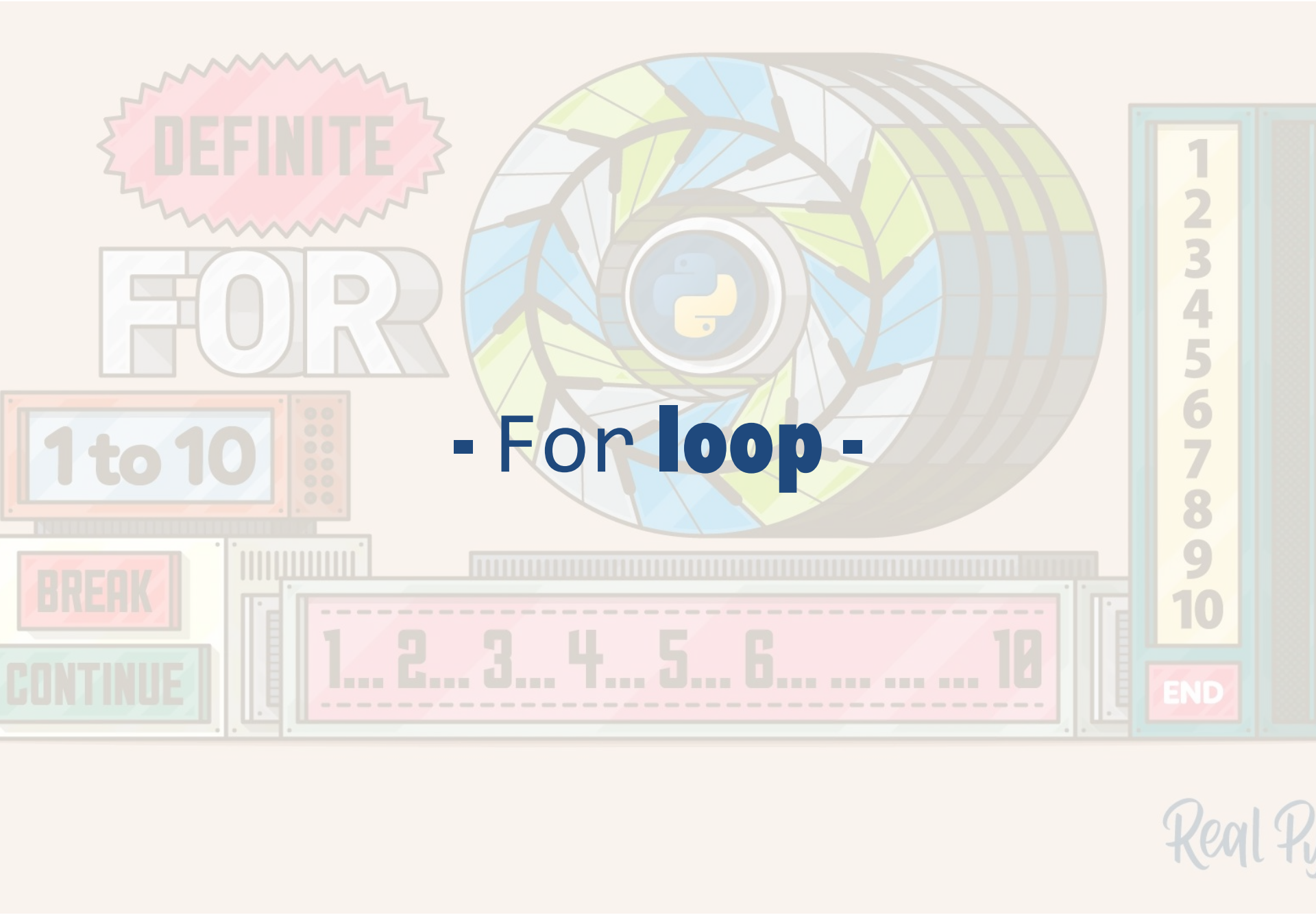
Indeed, it is a particularity of tuples. Let's see an example!

Tuples – an example

Let's assume you store medical data from various patients. In particular you will request the following information

Name: Joe
Age: 24
Weight: 65
Smoker: False

```
# (Name, Age, Weigth, Smoker)
patient_1 = ("Joe", 24, 65, False)
name = patient_1[0]
age = patient_1[1]
weight = patient_1[2]
smoker = patient_1[3]

# OR
name, age, weight, smoker = patient_1
```

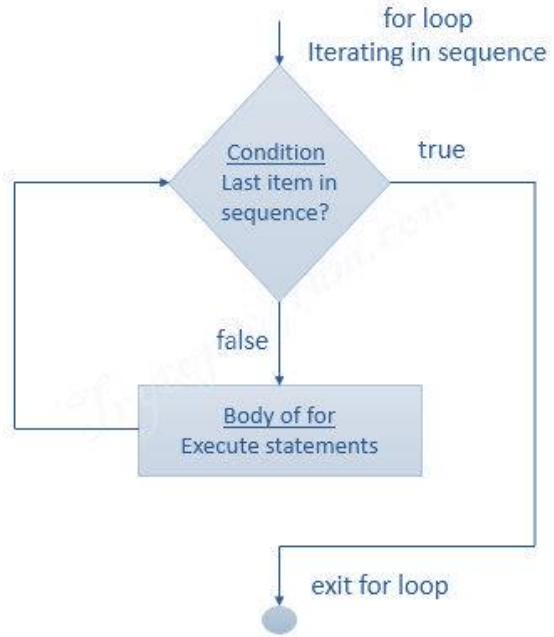

DEFINITE
FOR
1 to 10
BREAK
CONTINUE
1... 2... 3... 4... 5... 6... 10
1
2
3
4
5
6
7
8
9
10
END

- For **loop** -

For loop

! Terminology alert !

For loop: The FOR loop is a programming construct used to repeat a set of instructions for a fixed number of times. It iterates over a sequence of values, such as a list or a range, and performs the same set of instructions on each value.



Example

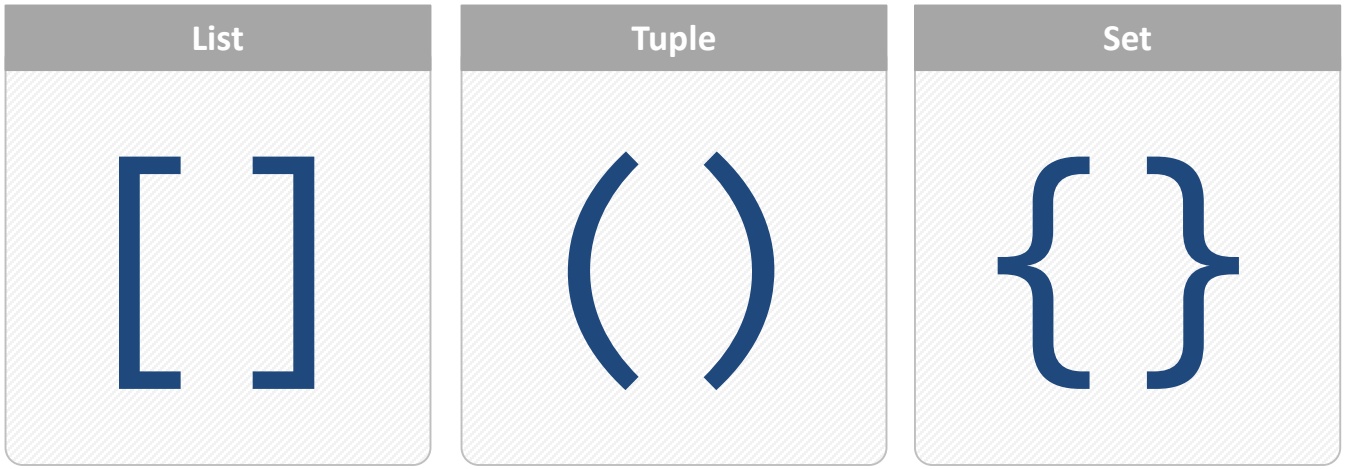
```
# Print the numbers 0 to 4
for i in range(5):
    print(i)
```

What is this code doing exactly?

```
0
1
2
3
4
```

For loop

In python for loops are designed to iterate through containers such as:



iterator

```
for x in something:
    # Code block
```

Iteration with for loop

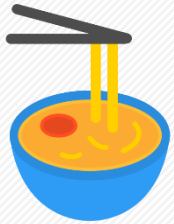
- After building a container, most applications need to do something with each item in it
- The idea is to "**step through**" the container to do something to each object
- This type of operation is called **iteration**

For example

To find the largest item in an (unsorted) list, an algorithm would need to check the value of every item during its search! This algorithm will be examined a little later

For loop: A delicious example

You are the chef a gourmet restaurant and you want to automatically print your menu


Ramyeon


Jjigae


Bibimbap


Tteokbokki

You also happened to be a talented programmer and you save your menu in a list

```
menu = ["Ramyeon", "Jjigae", "Bibimbap", "Tteokbokki"]
```

Now you want to print your menu automatically on a screen for your customers

For loop: A delicious example

Here is the code you would use to display one dish on each line

code

```
menu = ["Ramyeon", "Jjigae", "Bibimbap", "Tteokbokki"]

for dish in menu:
    print(dish)
```

output

```
Ramyeon
Jjigae
Bibimbap
Tteokbokki
```

Let's have a closer look to understand what is hapenning

Iteration variable: it represents all your elements once at a time

Begins the statement

indentation

```
for dish in menu:
    print(dish)
```

Name of the list you would like to loop over

Begins body of loop (in the next line)

Connected keyword

Body

Iteration variable

For loop: A delicious example

Let's use our debugging ability to see what is happening in this loop step-by-step

```
menu = ["Ramyeon", "Jjigae", "Bibimbap", "Tteokbokki"]
```

```
for dish in menu:
    print(dish)
```

Variable	Value
dish	'Tteokbokki'

Note that the statements inside a for loop – the body of the loop – must be indented

- Python assigns `dish` to be the first item in the list and then executes the indented statement(s)
- Then it gets the next item, assigns it to `dish`, and executes the indented statement(s) again
- It repeats until all the items in list have been processed

Note: this slide is intended to work with animation

For loop: A delicious example

We can also perform more computation in the loop

For instance we can display the number of letters per dish!

code

```
menu = ["Ramyeon", "Jjigae", "Bibimbap", "Tteokbokki"]

for dish in menu:
    word_length = len(dish)
    print(dish + " " + str(word_length))
```

output

```
Ramyeon 7
Jjigae 6
Bibimbap 8
Tteokbokki 10
```

Admittedly, the number of letters per dish is not very meaningful for the customer. Instead we can display their price

For loop: A delicious example

First we need to find the most convenient way to store the data

Here we can use a nested list

Now each dish is associate with its name and price.

Let's see how we iterate through it

```
for dish in menu:
    name = dish[0]
    price = dish[1]
    print(name + " " + str(price))
```

```
Ramyeon 6
Jjigae 12
Bibimbap 10
Tteokbokki 8
```

For loop: A delicious example

Let's see more in details what is happening in this loop!

```
menu = [["Ramyeon", 6], ["Jjigae", 12], ["Bibimbap", 10] , ["Tteokbokki", 8]]
```

```
for dish in menu:
    name = dish[0]
    price = dish[1]
    print(name + " " + str(price))
```

Variable	Value
dish	['Tteokbokki', 8]
name	'Tteokbokki'
price	8

Note: this slide is intended to work with animation. You can replicate it with your own trace

Output

For loop: A delicious example

Caution

If you change the value of the iteration variable you also modify its value in the container!!

For instance if you want to change all the price to 10$

```
menu = [["Ramyeon", 6], ["Jjigae", 12], ["Bibimbap", 10] , ["Tteokbokki", 8]]

for dish in menu:
    dish[1] = 10

print(menu)
```

output [['Ramyeon', 10], ['Jjigae', 10], ['Bibimbap', 10], ['Tteokbokki', 10]]

For loop: iteration through a string

A string is also an iterable!

So you can run a FOR loop through it

Code

```
name = "SUNY-Korea"
for letter in name:
    print(letter)
```

Output

```
S
U
N
Y
-
K
o
r
e
a
```

Note: A string is also an iterable!

For loop: iteration through a string

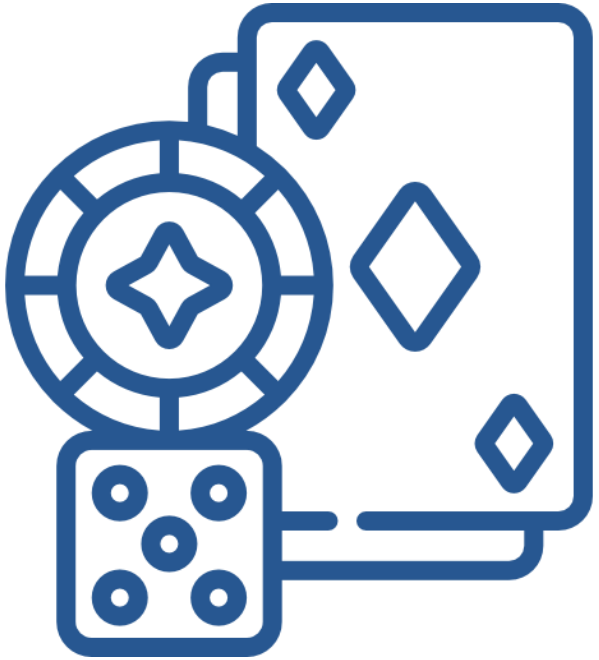
But it is NOT working for a numerical variable

Not iterable

Get back to our previous example

Let's build a gambling machine:

1. A random integer between 0 to 10 is generated
2. The player has to guess the number and can win the grand prize
3. A player has maximum 4 trials



How are we going to do that?

Gambling machine

One solution would be to create a list containing as many values as we want iterations

```
import random
# Generate a random integer between 1 and 10 (inclusive)
number_to_guess = random.randint(1, 10)
it_list = [0,1,2,3]
for i in it_list:
    print("### trial {} ###".format(i+1))
    user_number = int(input("enter a number between 0 to
10 : "))
    if (number_to_guess!=user_number):
        print("you loose")
    else:
        print("you win")
```

List of length 4 created solely to limit the number of iterations

But what if I want 10,000 iterations?

range()

For this case we use the function `range()`

In short

The function range is used to quickly create a list of integers!

For instance:

```
range(5)
```

```
[0, 1, 2, 3, 4]
```

```
range(10)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- **range(n)** means "the sequence of integers starting from zero and ranging up to, but not including, **n**"
- Python executes the body of the loop **n** times
- **i** is set to every value between 0 and **n-1** (**n** is NOT included)

Important note

Here I am simplifying a little bit, range generates integers on the fly and has its own class. So it is not strictly speaking a list, you can convert a range to a list with a cast

```
list(range(10))
```

range()

Now let's use range in a for loop

Let's try to print something n times

```
n = 3
for i in range(n):
    print("this text")
```

```
this text
this text
this text
```

Since the iteration variable is never explicitly used in the core of the loop we can specify it by changing its name by a _

Note that now I can make very long loops!

```
n = 10000
for _ in range(n):
    print("this text")
```

```
this text
this text
this text
```

range()

`range()` can also take multiple arguments to generate a list of interger in a specified range

```
for i in range(5,10):
    print(i)
```

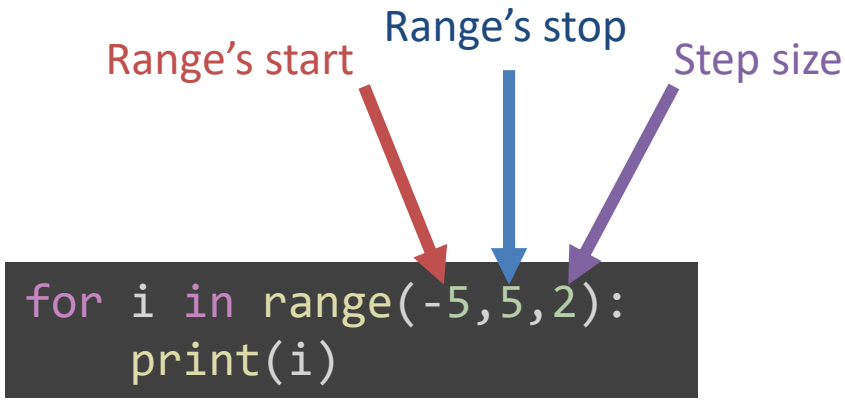
```
5
6
7
8
9
```

```
for i in range(-5,5):
    print(i)
```

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

range()

You can also adjust the step size




```
-5
-3
-1
1
3
```

Gambling machine using range

To implement this code, we will use a for loop

```
import random

# Generate a random integer between 1 and 10 (inclusive)
number_to_guess = random.randint(1, 10)

for i in range(4):
    print("### trial {} ###".format(i+1))
    user_number = int(input("enter a number between 0 to 10 : "))
    if (number_to_guess!=user_number):
        print("you loose")
    else:
        print("you win")
```

Thanks to the for loop, the function will quit after 4 trials

You adjust the number of trial by changing the values in `range()`

For loop
- Example -
Sum calculator

Example 1: Sum calculator

Consider a function that computes the sum of the numbers in a list

Note this function exist in Python, named `sum()`, but by thinking how to write it, we can better understand for loops

What do we want to do?

A list → [1, 2, 3, 5, 7]

The sum of all elements → 1+2+3+5+7

Implementation step-by-step:

- First, initialize a variable **total** to zero
- Then, use a **for loop** to add each number in the list to **total**
- After all items have been added, the loop will terminate, and the function returns the final value of **total**

Example 1: Sum calculator

Let's understand each part of this code!

Initialize a variable to store the running total

Visit each number in the list of numbers

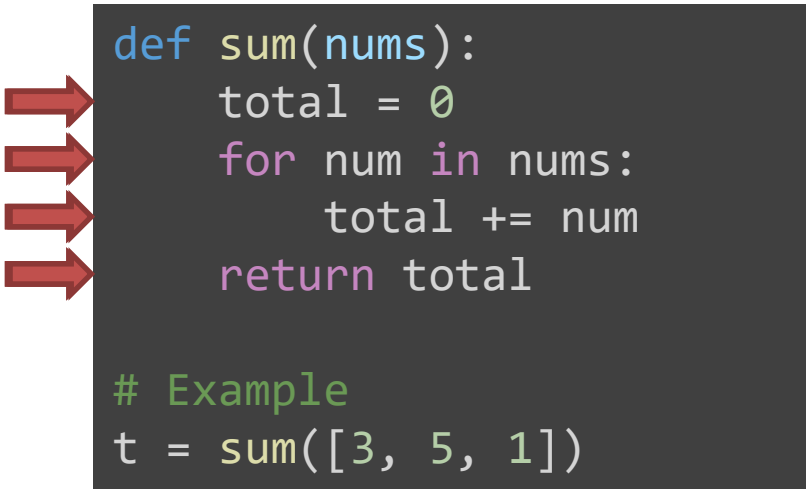
Add each number to the running total

Return the final total

```
def sum(nums):
    total = 0
    for num in nums:
        total += num
    return total

# Example

t = sum([3, 5, 1])
```

Example 1: Sum calculator

Now we will trace this code to understand it better



```
def sum(nums):
    total = 0
    for num in nums:
        total += num
    return total

# Example
t = sum([3, 5, 1])
```

Variable	Value
total	9
num	1

For loop

Iteration using list indexes

Iteration using list indexes

We have seen that you can iterate through a list using

```
cars = ['Kia', 'Peugeot', 'Hyundai', 'Ford']
for car in cars:
    print(car)
```

We also know that we can access the elements in the list via their indexes

```
'Kia' == cars[0]
'Peugeot' == cars[1]
```

Then we can use the indexes from range to iterate through the list

```
cars = ['Kia', 'Peugeot', 'Hyundai', 'Ford']
for i in range(len(cars)):
    print(cars[i])
```

Iteration using list indexes

This function computes and returns the sum of the first k values in a list

Generate indexes 0 to *k*-1

Add each number to the total

Return the final total

```
def partial_total(nums, k):
    total = 0
    for i in range(k):
        total += nums[i]
    return total
```

```
# Examples
a = [4, 2, 8, 3, 1]
partial_total(a, 3)  # returns the value 14
partial_total(a, 1)  # returns the value 4
partial_total(a, 6)  # error
```

Iteration using list indexes

Trace execution

```
def partial_total(nums, k):
    total = 0
    for i in range(k):
        total += nums[i]
    return total
```

Variable	Value
total	14
i	2
nums[i]	8

```
# Examples
a = [4, 2, 8, 3, 1]
partial_total(a, 3)
```

Iteration using list indexes

You can find many usage where you would prefer to use list indexes instead of iterating through a list

Iterate through multiple lists at once

```
cars = ['Kia', 'Peugeot', 'Hyundai', 'Ford']
prices = [15000, 18000, 20000, 31000]
```

Imagine you would like to iterates through these two lists simultaneous without creating a more complex datastructure

```
cars = ['Kia', 'Peugeot', 'Hyundai', 'Ford']
prices = [15000, 18000, 20000, 31000]
for i in range(len(cars)):
    print("A " + cars[i] + " car will cost you " \
        + str(prices[i]) + "$")
```

```
A Kia car will cost you 15000$
A Peugeot car will cost you 18000$
A Hyundai car will cost you 20000$
A Ford car will cost you 31000$
```

Note: In this particular use case you might want to use enumerate or another container instead

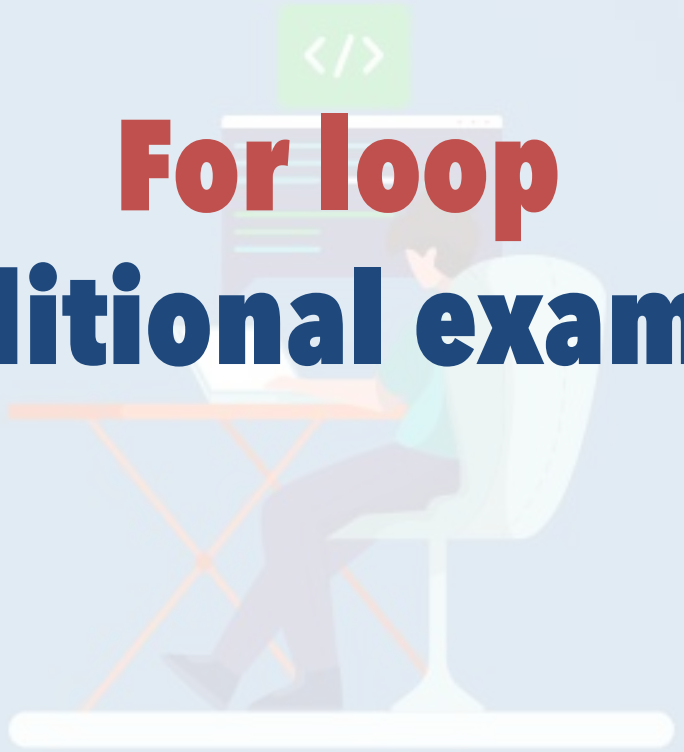
String iteration using indexes

We have seen strings are iterable!

```
for s in range(len(my_string)):
    print(…)    # What should be added here to display each letter??
```

For loop

- Additional examples -



- Example 1 - Maximum

Find maximum in a list

Let's write an algorithm to find the maximum value in a list

Basic idea: **iterate over the list and keep track of the larger value seen to that point**

- Begin by taking the value at index 0 as the maximum
- Continue with the remainder of the list, comparing the next value with the current maximum and updating the maximum if and when a larger value than the current maximum is found

```
def find_max(nums):
    maximum = nums[0]
    for i in range(1, len(nums)):
        if nums[i] > maximum:
            maximum = nums[i]
    return maximum

ages = [20, 16, 22, 30, 17, 24]
max_age = find_max(ages) # max_age will be 30
print('Maximum age: ' + str(max_age))
```

Find maximum in a list

```
def find_max(nums):
    maximum = nums[0]
    for i in range(1, len(nums)):
        if nums[i] > maximum:
            maximum = nums[i]
    return maximum

ages = [20, 16, 22, 30, 17, 24]
max_age = find_max(ages) # max_age will be 30
print('Maximum age: ' + str(max_age))
```

Variable	Value
maximum	30
i	5
nums[i]	24

- Example 2 -
Count vowels

Count vowels

A for loop can iterate over the characters in a string

- To see how this works, consider a function called **`count_vowels`** that counts the number of vowels (lowercase or uppercase) in a word:
  - To make this problem a little easier to solve, we can call the **`lower()`** method for strings, which makes a copy of a given string and changes all the uppercase letters to lowercase
  - **`upper()`** makes all letters uppercase
- Strings are **immutable** (unchangeable) objects
- To convert a string into lowercase we must make a lowercase copy of it and replace the original string with the new one

Count vowels

```
def count_vowels(word):
    vowels = 'aeiou'
    num_vowels = 0
    for letter in word.lower(): # search through a
        if letter in vowels:            # lowercase copy of
            num_vowels += 1         # the original word
    return num_vowels

word = 'Cider'
print('The number of vowels in ' + word + ' is ' +
str(count_vowels(word))) # will print 2
```

Variable	Value
num_vowels	2
letter	r

- Example 3 – Average

Average

In this exercise, you are a professor. You have a group of **4** students and for each, there are **3** exam scores.

	Exam 1	Exam 2	Exam 3
Student 1	89	85	90
Student 2	78	85	72
Student 3	99	86	92
Student 4	82	84	79

It can be written in python as follows

```
scores = [  [89, 85, 90],
            [78, 85, 72],
            [99, 86, 92],
            [82, 84, 79]]
```

- To access a particular score, two indices are needed:
  - First, which students grade is needed (0 through 3)
  - Second, which score of that student is desired (0 through 2)
- **Example**: `scores[3][1]` is fourth student's score on the second exam (which is 84)

Average

We want to write code that will compute the average score that students earned on each exam

We will write more than one version of the program ➔ But start simple

Version 1

In the first version, we will "hard-code" several values (the number of students and the number of scores) in the program

Version 2 & more

Then, generalize things a bit and use variables for these values

Average version 1

```
# scores represents 4 students who each took 3 exams
scores = [[89, 85, 90], [78, 85, 72],
          [99, 86, 92], [82, 84, 79]]

averages = [0, 0, 0]

for student in scores:
    averages[0] += student[0]
    averages[1] += student[1]
    averages[2] += student[2]

for i in range(3):
    averages[i] /= 4

print(averages)
```

Average version 2

Limitations:

- The first version of the code has a major drawback: the algorithm will work only for a class of four students who took three exams
- Suppose the class is larger or smaller? Or suppose the students took more or fewer exams?

What can we do?

- Next development attempt is a better (but more complicated) version of the algorithm that can adapt to larger/smaller class sizes and more/fewer exams
- The approach will rely on **nested loops**, which means there will be one loop inside of another
- Nested loops will become increasingly important as the course progresses

Average version 2

A trick to initialize vectors

- One other thing before looking at the program
- Recall that syntax like **'Hi'*3** will create a new string by repeating a given string a desired number of times
  - For instance, **'Hi'*3** equals **'HiHiHi'**
  - In a similar manner, **[0]*3** would create a list containing 3 zeroes, namely, **[0, 0, 0]**
- Thus, the * notation with strings and lists is essentially a form of **concatenation**

Average version 2

```
scores = [[89, 85, 90], [78, 85, 72], [99, 86, 92], [82, 84, 79]]

num_students = len(scores)
num_exams = len(scores[0])      # each student took the
averages = [0] * num_exams      # same number of exams

for student in scores:
    for i in range(0, num_exams):       # nested loops
        averages[i] += student[i]

for i in range(0, num_exams):
    averages[i] /= num_students

print(averages)
```

Average version 3

- In a third and final version of the exam average calculator, the computations will be *encapsulated* (enclosed or wrapped) inside of a function **`compute_averages(students)`**

- The function takes the list of scores as its argument

- After computing the exam averages, the function returns a list of the average scores

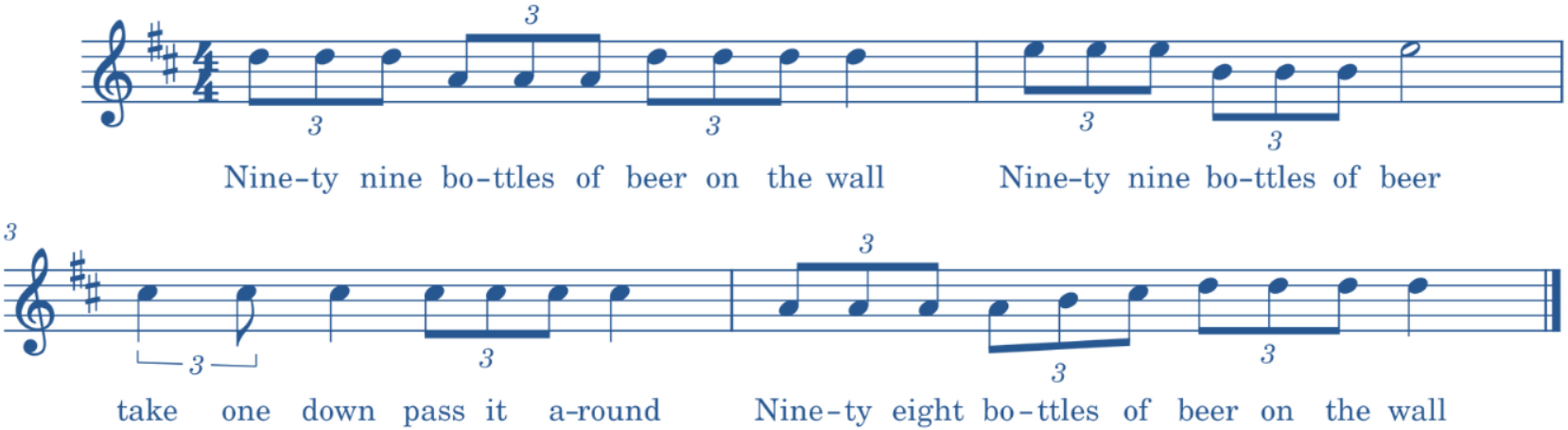
- This illustrates that Python functions can return many values at once (via a list), not just a single number or string

Average version 3

```
scores = [[89, 85, 90], [78, 85, 72], [99, 86, 92], [82, 84, 79]]

def compute_averages(students):
    num_students = len(students)
    num_exams = len(students[0])
    avgs = [0] * num_exams

    for student in students:
        for i in range(0, num_exams):
            avgs[i] += student[i]

    for i in range(0, num_exams):
        avgs[i] /= num_students

    return avgs

averages = compute_averages(scores)
print(averages)
```

- Example 4 –
Bottle of beer

99 bottles of beer on the wall



- The final example is on a lighter note looking at a program that prints the lyrics of the song "99 Bottles of Beer on the Wall"
  - In this song, the singer needs to count from 99 down to 0
- The **`range`** command can be used to count up, but it also can count down if given a negative number for the step size
- For example, **`range(10,-1,-1)`** will count down from 10 to 0 by 1s
- So **`list(range(10,-1,-1))`** would generate the list [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
- The code on the next slide asks the user for the starting number so that the program can start from a value other than 99

99 bottles of beer/milk on the wall

```
age = int(input('How old are you? '))

if age < 21:
    drink_type = 'milk'
else:
    drink_type = 'beer';

num_bottles = int(input('How many bottles of ' + drink_type + ' do you have? '))

for bottle in range(num_bottles, -1, -1):
    if bottle > 1:
        print(str(bottle) + ' bottles of ' + drink_type +
              ' on the wall!')
    elif bottle == 1:
        print('1 bottle of ' + drink_type + ' on the wall!')
    else:
        print('No bottles of ' + drink_type + ' on the wall!')
```

Some practices

Vowels - Improvements

1. Also count and return the number of non-vowel letters
– Hint: use a list to return both numbers
2. Print out the number of vowels and non-vowels
- Hint: need to access the index of the returned tuple

Bottle of beer - Improvements

1. Write milk only for people younger than the legal drinking age in your country
2. Make it count up from 1 to the user input number, incrementing by 2 (e.g. 1,3 ,5…)

Let's address a bigger challenge - the sieve of Eratosthenes -

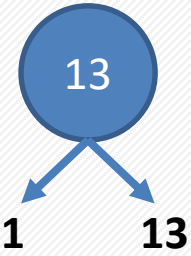
Prime numbers

What is a prime number?

! Terminology alert !

A **prime number** is a positive integer greater than 1 that can only be divided evenly by 1 and itself.

Examples



13 has only two factors – itself and 1. So it is a prime number

4 has three factors – itself, 1 and 2. So it is NOT a prime number

Prime numbers have many implications in cryptography, number theory, mathematics, computing and more

The Sieve of Eratosthenes

Prime numbers have been a topic of interest for mathematicians for **centuries**

Eratosthenes

- The ancient Greek mathematician, Eratosthenes, was one of the first known mathematicians to study prime numbers
- Eratosthenes developed a **sieve** method to identify prime numbers, now known as the Sieve of Eratosthenes
- This method is still in use nowaday

Born in 276BC

Interesting note

Other famous mathematicians contributed to the study of prime numbers such as Euclid, Euler and Gauss.

The Sieve of Eratosthenes

The Sieve of Eratosthenes can find a list of primes up to some limit N

For instance, let's find all the primes between 0 to 100

Note that 1 is not prime and is not included in the list

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

If we want to calculate all primes to 200 it will work in the same fashion

The Sieve of Eratosthenes

We start at the beginning, if we find an unmarked number. We keep it as a **prime**

2 is prime because it is unmarked!

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

The Sieve of Eratosthenes

Now we can eliminate all multiples of 2 because we know they are composite

2 is prime because it is unmarked!

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

The Sieve of Eratosthenes

The move to the next non-marked number! 3 is unmarked, so we set 3 as a **prime number**

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

The Sieve of Eratosthenes

Notice that 6 has already been marked! It is actually unnecessary to check the number $< 3^2$

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

The Sieve of Eratosthenes

Then we mark all the multiples of 3 as composite numbers

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

The Sieve of Eratosthenes

We then move to the next un-marked number. In this case it is the 5 which is indeed a **prime number**!

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

The Sieve of Eratosthenes

And we mark all the multiple of 5 as composite numbers

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

The Sieve of Eratosthenes

We then move to the next unmarked number in the list. This time it is 7, which is a prime number

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

The Sieve of Eratosthenes

We mask all the multiples of 7

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

The Sieve of Eratosthenes

We now move to 11

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

The Sieve of Eratosthenes

Notice that 11*11 is greater than 100 so we can stop! All the remaining numbers are primes

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

The Sieve of Eratosthenes

Notice that 11*11 is greater than 100 so we can stop! All the remaining numbers are primes

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

The Sieve of Eratosthenes

The method depicted previously works well and you can even apply it by hand

But what if prime numbers between 2 and 10,000 are needed?

- **It is a tedious process to write a list of 10,000 numbers**
- **And it is likely you will do some arithmetic mistakes in the process**

Can this method be turned into a computation?

Yes, but we need to carefully design our strategy and divide it into smaller and more manageable pieces

Thinking of the problem

Before you start coding, you need to think how to solve sub-parts of the problem

For example

Create a list?

Marking the composites?

Stopping criterion?

How to divide in functions?

Initializing the list

First of all, we need to declare a list containing the numbers we would like to process

Range The user will have to define the range in which he would like to work

MIN_NUMBER = 2 max_number = 2

```
max_number = 100
MIN_NUMBER = 2
```

Create your list

```
list(range(MIN_NUMBER,max_number))
```

Cast into list

Lower limit

upper limit

Initializing the list

Is the previously generated list satisfying?

Print the list

```
print(list(range(MIN_NUMBER,max_number)))
```

```
[2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 2
4, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43,
44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63,
64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83,
84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99]
```

The list stops at n-1

The make list between 2 and 100:

```
list(range(MIN_NUMBER,max_number+1))
```

Masking composites

When we will iterate through our list, we will have to "remove"/mark the composite numbers. How would you do it?

Using list methods? Multiple list methods are designed to remove elements from a list

`list.remove(x)`
Remove the first item from the list whose value is equal to *x*. It raises a `ValueError` if there is no such item.

`list.pop([i])`
Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. (The square brackets around the *i* in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)

`list.clear()`
Remove all items from the list. Equivalent to `del a[:]`.

These approaches cannot be used during iterations through the object!

Instead we can replace the composite number which a recognizable value?

None

```
my_list = list(range(1,5))
print(my_list)
my_list[2] = None
print(my_list)
```

```
[1, 2, 3, 4]
[1, 2, None, 4]
```

Initializing the worksheet

As for now, our list starts from 2 and ends at 100

Would it be convenient to manipulate?

Let's call the list on which we will work worksheet

```
worksheet = list(range(MIN_NUMBER,max_number+1))
```

It might simplify our strategy if the index and the value coincide!

What about placing two placeholder first to represent 0 and 1?

```
worksheet = [None, None] + list(range(MIN_NUMBER,max_number+1))
```

Or

```
worksheet = list(range(MIN_NUMBER,max_number+1))
worksheet.insert(0,None)
worksheet.insert(0,None)
```

Or

```
worksheet = list(range(0,max_number+1))
for i in range(0,2): worksheet[i]=None
```

Iterating through the worksheet

Now that you have your list, you know that you will need a strategy to visit the elements inside it

We will use a FOR loop

```
for i in range(0, max_number+1):
    print(i)
```

➡

```
0
1
2
3
⋮
98
99
100
```

Remember that we have to care only about **non-masked elements** (None)

```
for i in range(0, max_number+1):
    if (worksheet[i] != None):
        print(i)
```

```
2
3
⋮
98
99
100
```

Iterating through the worksheet

Is the loop we made before good?

Well … it will work but it is not optimal, we can have a better stopping criterion

```
for i in range(2, int(math.sqrt(max_number))):
    if (worksheet[i]!=None):
        print(i)
```

Instead of the `print`, we want a code to find the multiples of the current index `i` and mask them in the list

Let's create a function!

Since this part will be called respectively in the code and to simplify the development, we will create a function!

Let's call this function to `sift` that will remove the multiple of a specific number in a list!

Sift function I/O

How our function sift will look like?

First! Let's think about the inputs and outputs of the function

Input

1. The "worksheet"
2. The current selected prime

output

1. The modified worksheet

BUT WAIT!!

Do you remember what is happening when we use the operator = to copy a list??

What is happening if they modify a list in function??

The Sieve of Eratosthenes

Remember about local variables?

```
def my_function(a):
    a = 3
    return a
a = 2
print(a)
my_function(a)
print(a)
```

```
2
2
```

Changing the local variable does not modify the global variable

What about a list?

```
def my_function(list_a):
    list_a[1] = 100

list_a = [1, 2, 3]
print(list_a)
my_function(list_a)
print(list_a)
```

```
[1, 2, 3]
[1, 100, 3]
```

In Python, when a list is passed to a function as an argument, it is passed by reference rather than by value. This means that the function does not receive a copy of the original list, but rather a reference to the same list object in memory.

Sift function definition

Let's write our function definition

```
def sift(numbers, current_value):
```

The list of numbers (worksheet)

Current prime value

The function will return nothing because we can directly modify the list inside, and it will also modify the global variable!

Sift function

Remember that in this function, we need to find all the multiples of the selected prime (`current_value`) and mask them in the list (with None)

```
def sift(numbers, current_value):
    max_val = len(numbers)+1
    for i in range(current_value*current_value, max_val, current_value):
        numbers[i] = None
```

Any multiple of `current_value` will be replaced by a None

Step-size

The Sieve of Eratosthenes: almost!

Let's use this function in our main loop!

```
import math

# function sift checking
def sift(numbers, current_value):
    max_val = len(numbers) + 1
    for i in range(current_value*current_value, max_val, current_value):
        numbers[i] = None

# Declare the list of number
#  Add two first elements to simplify the indexing
max_number = 100
MIN_NUMBER = 2
worksheet = [None, None] + list(range(MIN_NUMBER,max_number+1))

# Run the main loop
for i in range(2, int(math.sqrt(max_number))):
    if (worksheet[i]!=None):
        sift(worksheet, i)
```

[None, None, 2, 3, None, 5, None, 7, None, None, None, 11, None, 13, None, None, None, 17, None, 19, None, None, None, 23, None, None, None, None, None, 29, None, 31, None, None, None, None, None, 37, None, None, None, 41, None, 43, None, None, None, 47, None, None, None, None, None, 53, None, None, None, None, None, 59, None, 61, None, None, None, None, None, 67, None, None, None, 71, None, 73, None, None, None, None, None, 79, None, None, None, 83, None, None, None, None, None, 89, None, None, None, None, None, None, 97, None, None, None]

The nonNone function

Now we simply need to remove the None from the list and we are DONE!

- We can make a new helper function called **nonNone** returns a copy of the worksheet, but without any **None** objects
- It makes an initial empty list named **res** (for "result")
- Then it uses a for loop to look at every item in the input list
- If an item is not **None**, the item is appended to **res** using the **append** method for lists
- When the iteration is complete, **res** is returned as the result of the function call

The nonNone function

Initialize res to be an empty list

Visit each element in the list numbers

See if val is actually a number

If val is a number, then append it to res

```
def nonNone(numbers):
    res = []
    for val in numbers:
        if val is not None:
            res.append(val)
    return res
```

```
# Usage
worksheet = [None, None, 2, 3, None, 5, None, 7, None, None,
                          None, 11, None, 13, None, None]
worksheet = nonNone(worksheet)    # worksheet is now: [2, 3, 5, 7, 11, 13]
```

The Sieve of Eratosthenes: Done!

Here is the final version of the Sieve of Eratoshthenes

```
import math

def sift(numbers, current_value):
    max_val = len(numbers) + 1
    for i in range(current_value*current_value, max_val, current_value):
        numbers[i] = None

def nonNone(numbers):
    res = []
    for val in numbers:
        if val is not None:
            res.append(val)
    return res

def sieve(max_number):
    MIN_NUMBER = 2
    worksheet = [None, None] + list(range(MIN_NUMBER,max_number+1))
    for i in range(2, int(math.sqrt(max_number))):
        if (worksheet[i]!=None):
            sift(worksheet, i)
    return nonNone(worksheet)

# Run the cod
print(sieve(100))
```

Abstraction

- Now we have a function for making lists of prime numbers, which can be saved and used later
- It can be used to answer questions about primes, such as:
  - How many primes are less than n?
  - What is the largest gap between successive primes?
  - What are some twin primes (two prime numbers that differ only by 2, like 17 and 19)?
- This is a good example of **abstraction**: There is a nice, neat package that can be saved and **reused**
- In the future, there is no need to worry about the implementation details of sieve: just use it!
  - Just need to know that **sieve(n)** makes a list of prime numbers from 2 to n

Practice problems

Practice problem I

1. Write a function `is_even()` that returns True if n is an even number, and False otherwise

- First try to write it in whichever way you feel is most comfortable to you
- Then try to provide a one-line function body (optional)

2. Write a function `sum_even(lst)` that return the summation of all the even numbers in '`lst`'

- E.g. if lst is [4, 2, 6, 3, 1], then the return value should be 4+2+6=12
- Try using the is_even() function you wrote above, then try writing it without using is_even()

3. Write a function `remove_vowels(s)` that receives a string 's' and returns another string that does not contain the vowels

- E.g., if 's' is "I ate a banana", the return value should be " t bnn"
- Try to solve this after today's lecture

Practice problem II

Quick recap on accessing values via index

- Often an item in the middle of a list is needed
- If a list has n item, the locations in the list are numbered from 0 to n-1 (not 1 through n)
- The notation **a[i]** stands for "the item at location i in list a"

- In programming, use the word **index** to refer to the numerical position of an element in a list
- Example: **scores = [89, 78, 92, 63, 92]**

  scores[0] is 89

  scores[2] is 92

  scores[5] gives an "index out of range" error (why?)

Practice problem II

The **index method** will indicate the position of an element in a list

Official python documentation

`list.index(x[, start[, end]])`

Return zero-based index in the list of the first item whose value is equal to *x*. Raises a `ValueError` if there is no such item.

The optional arguments *start* and *end* are interpreted as in the slice notation and are used to limit the search to a particular subsequence of the list. The returned index is computed relative to the beginning of the full sequence rather than the *start* argument.

Example

```
scores = [89, 78, 92, 63, 92]
print(scores.index(92))
print(scores.index(99))
```

Print 2, the index of the first occurrence of 92 in the scores list

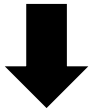
Generate this error: "ValueError: 99 is not in list"

Practice problem II

Important tip

If the program needs the index of a value, and it is not guaranteed the value is in the list, use an if statement in conjunction with the **in** operator to first make sure the item is actually in the list.

```
vowels = ['a', 'e', 'i', 'o', 'u']
letter = 'e'
if letter in vowels:
    print('That letter is at index ' + str(vowels.index(letter)) + '.')
else:
    print('That letter is not in the list.')
```



Output: **That letter is at index 1.**

Practice problem II

Problem 1: Implement the `index()` function on your own

- `def index(n, lst):`
- Returns the first occurrence of 'n' in '`lst`'
- What about the error cases?

Problem 2: Implement 'in' keyword

- `def is_in(n,lst)`
- Should give the same result as '`n in lst`'