

CSE 101

Computer Science Principle

Lecture 04: Iterations, Lists & Algorithm design

March. 2023

Prof. Francois Rameau

francois.rameau@sunykorea.ac.kr



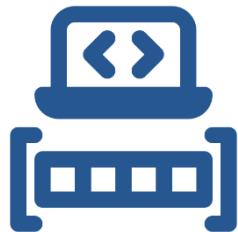
Attendance time!



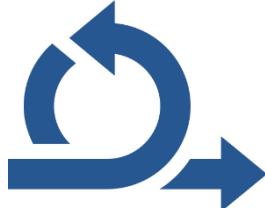
Scan the QR code and select your name in the list

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.

INDEFINITE

WHILE

$n > 0$

BREAK

CONTINUE

TRUE! TRUE! TRUE! TRUE!

Rea

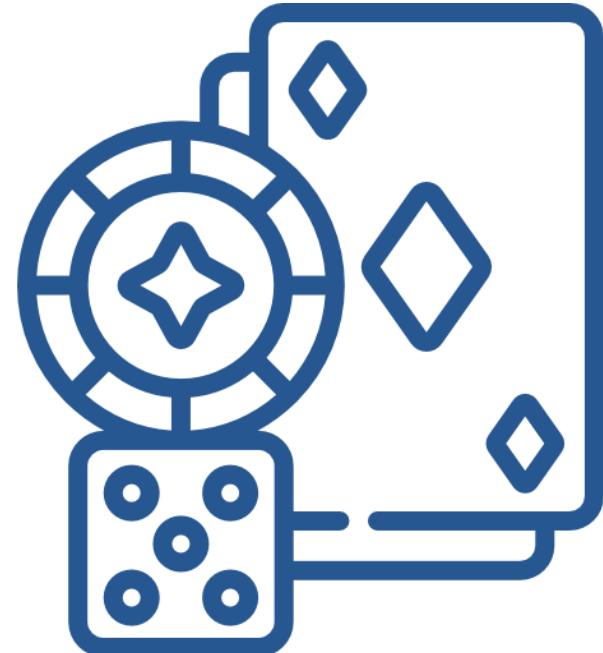
Loops and iterations

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 game:

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

```
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

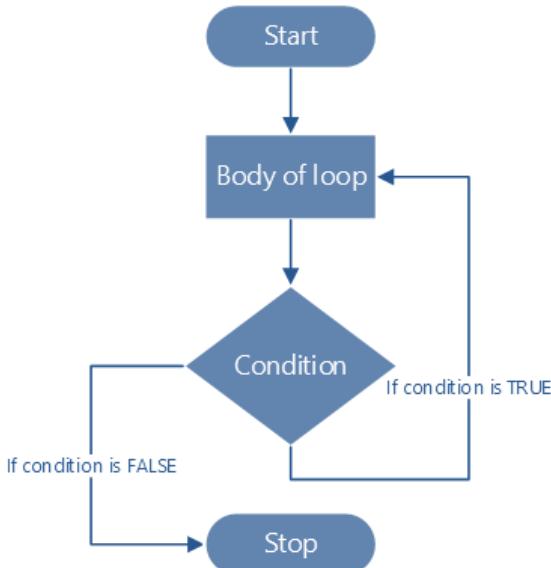


- While loop -

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

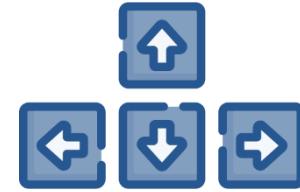
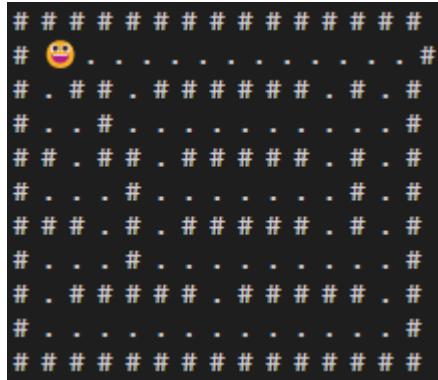
Let's say you now want to create a system collecting data, the user has to enter a few personal information. But you want your system to be able to detect unintended inputs

```
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 {}---".format(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 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
```

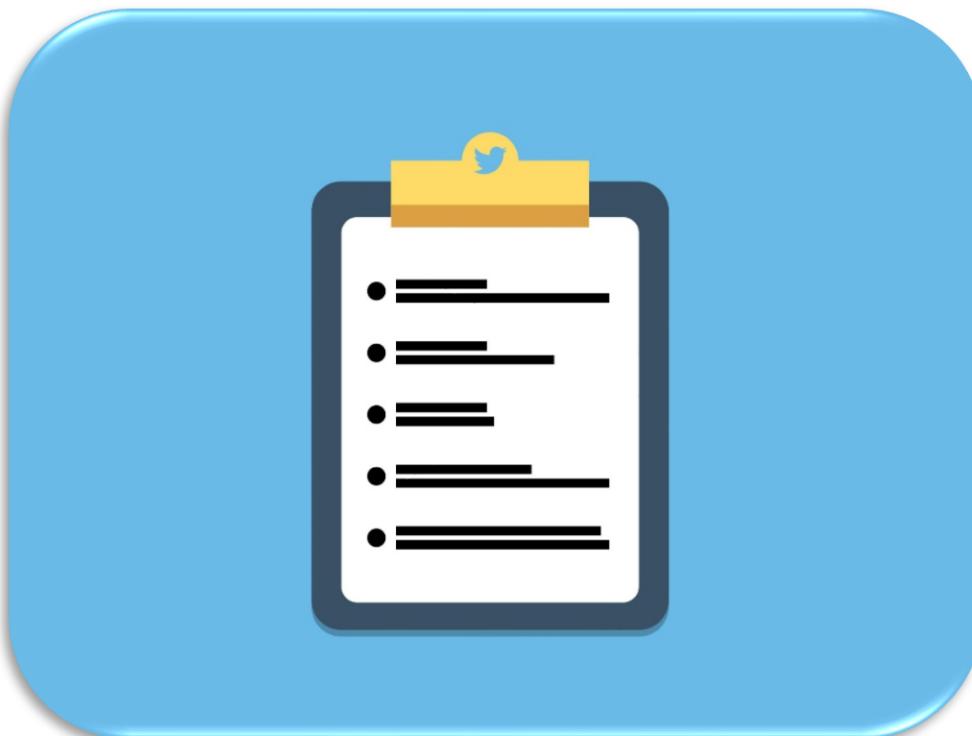
But if you do that, you need to take care of two things:

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

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

Missing ":"

```
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

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



Goal: You want to count from 10 to 0

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
```

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

1

2

Debugging

Logic errors 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 has tools to help you 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 errors in the 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?

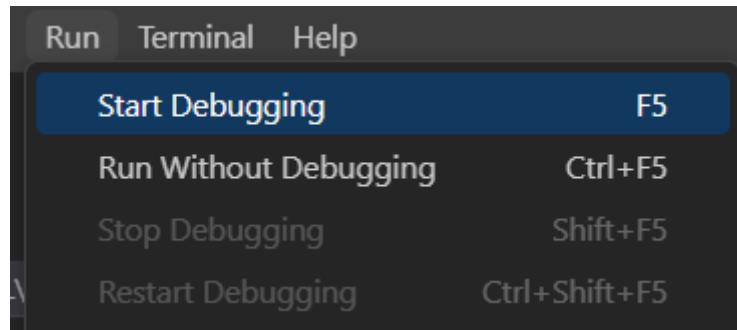
1. Set a breakpoint by clicking on the line number

```
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 !

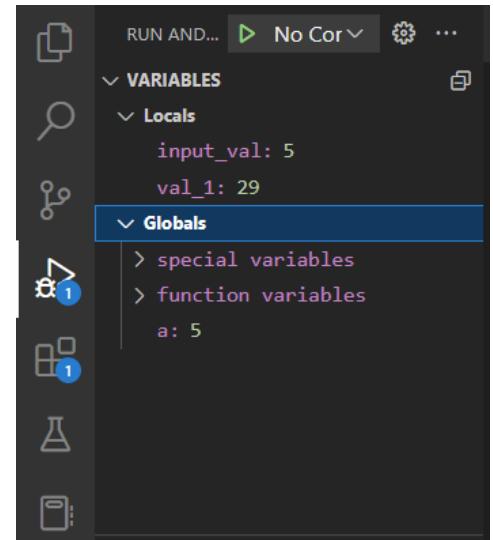


3. The code will stop at your breakpoint

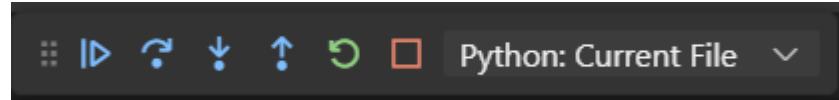
```
1  def function_1(input_val):  
2      val_1 = input_val**2 +4  
3      val_2 = function_3(val_1)  
4      return val_2  
5  def function_2(input_val):  
6      return input_val**0.5  
7  def function_3(input_val):  
8      val_1 = input_val/25  
9      val_2 = function_2(val_1)  
10     return val_2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



```
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)
```

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



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(step)`

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(next)`

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(return)`

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(list) [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`.

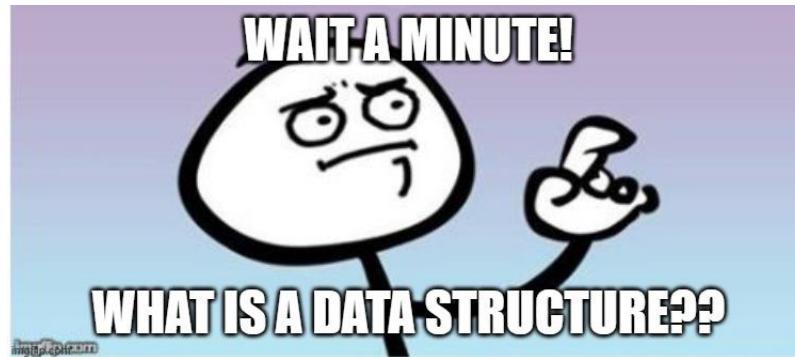
- Lists -

Inspired by:
[Socratica – YouTube](#)

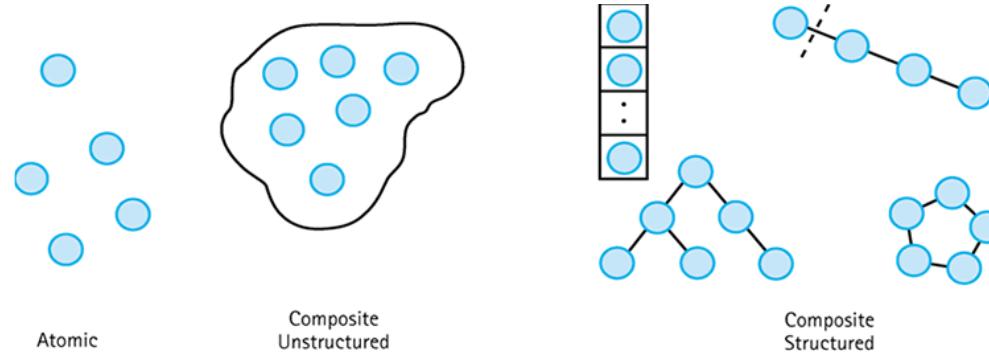
Real

Data structure

We will now see our first data structure!

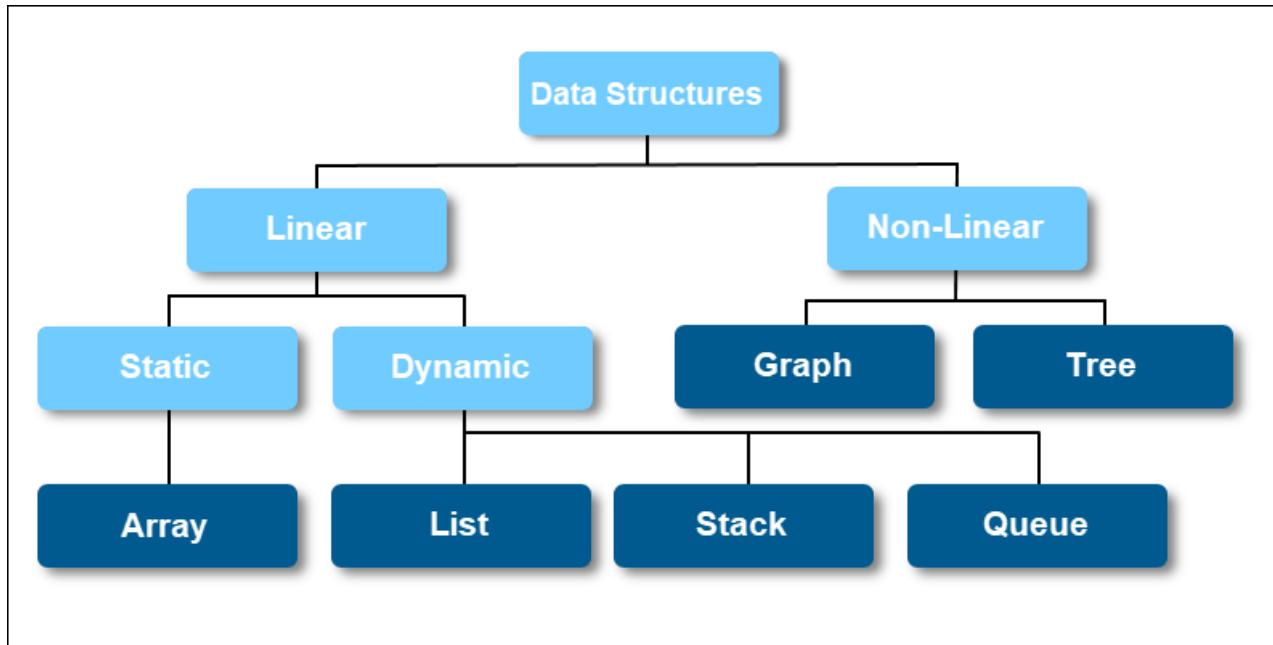


Data structure = A way of organizing data



Data structure

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



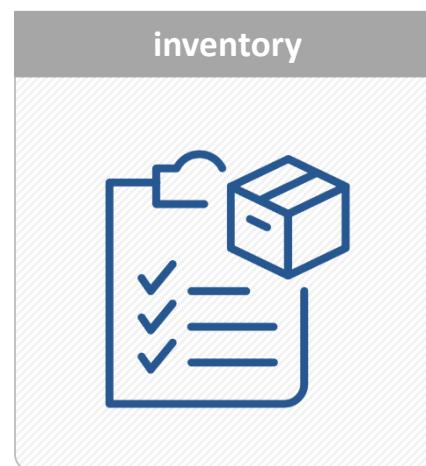
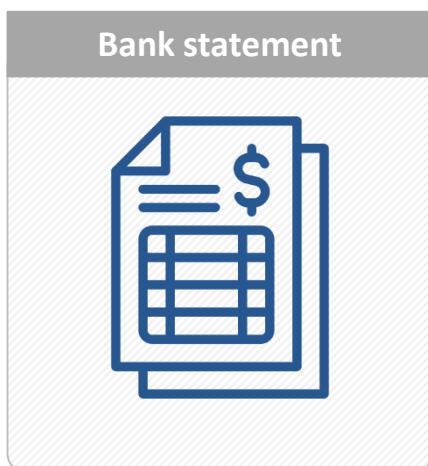
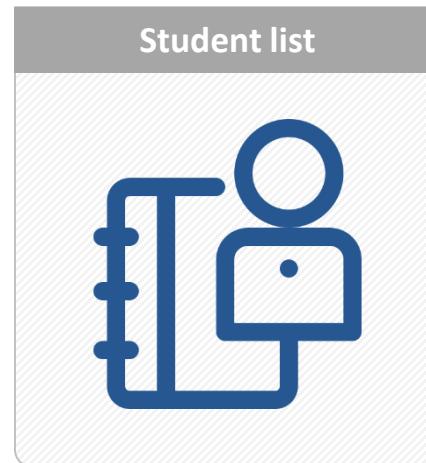
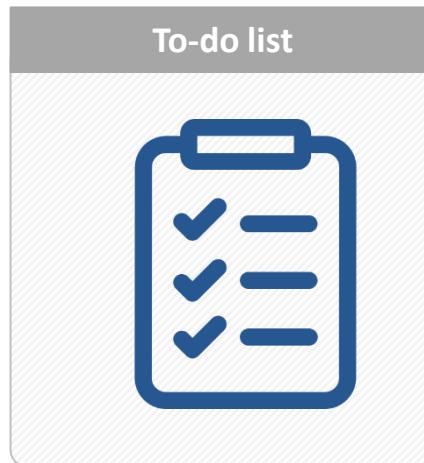
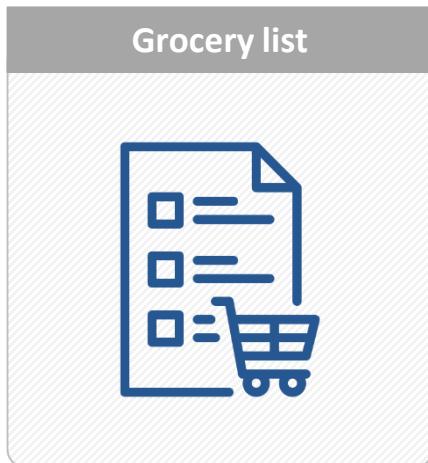
! Terminology alert !

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



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]
```

string

int

float

bool

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

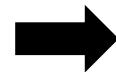
```
numbers = [1, 2, 3]
text = ['one', 'two', 'three']
print(numbers + text)
```



```
[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.
```

```

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 occurrences ---")
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 occurrence of a specific value---")
numbers.remove(3)
print(numbers)
# 8. reverse() --> 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

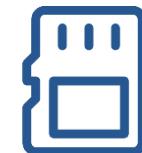


Tuples support much less methods than list!

So why do we use them??



Speed



Memory

Tuples

How to declare a tuple?

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

How do you access the elements inside?

Exactly like in a list!

<code>scores[1]</code>	60
<code>letters[2]</code>	'c'
<code>matrix[0][1]</code>	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, Weight, 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

- For loop -



1 ... 2 ... 3 ... 4 ... 5 ... 6 10

1 2 3 4 5 6 7 8 9 10

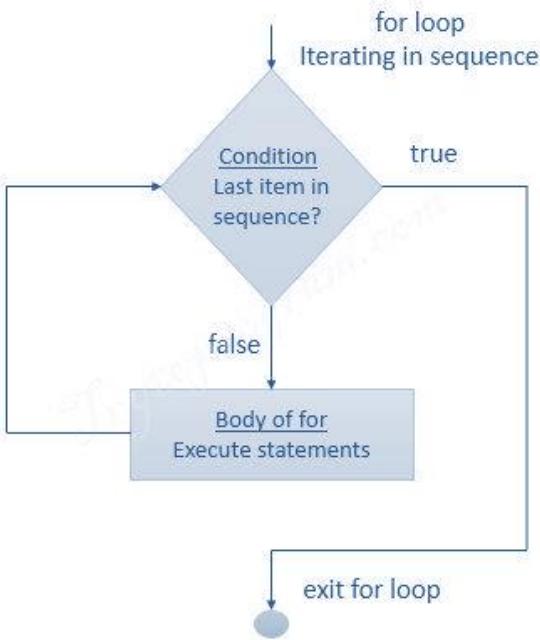
END

Real Py

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

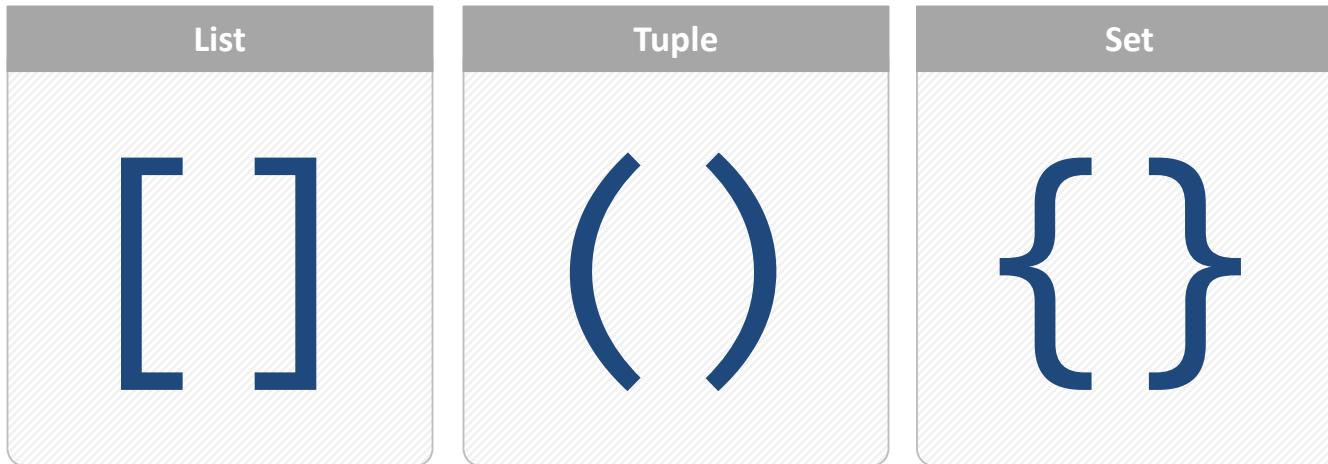
```
# Print the numbers 0 to 4
for i in range(5):
    print(i)
```

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:



for x in something:
 # Code block

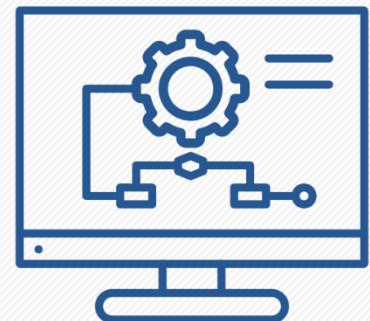
iterator

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)
```

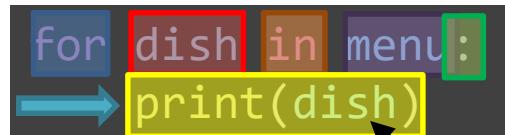
output

```
Ramyeon  
Jjigae  
Bibimbap  
Tteokbokki
```

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

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

Connected keyword
Body

Begins body of loop (in the
next line)

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

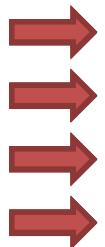
```
for dish in menu:  
    name = dish[0]  
    price = dish[1]  
    print(name + " " + str(price))
```

```
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!

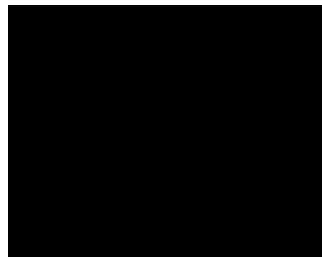
```
menu = [[ "Rameyeon", 6], [ "Jjigae", 12], [ "Bibimbap", 10] , [ "Tteokbokki", 8]]
```



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

Variable	Value
dish	[‘Tteokbokki’, 8]
name	‘Tteokbokki’
price	8

Output



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

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 = [[ "Ramen", 6], [ "Jjigae", 12], [ "Bibimbap", 10] , [ "Tteokbokki", 8]]  
  
for dish in menu:  
    dish[1] = 10  
  
print(menu)
```

output `[['Ramen', 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

Output

S
U
N
Y
-
K
o
r
e
a

Code

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

Note: A string is also an iterable!

For loop: iteration through a string

But it is NOT working for a numerical variable

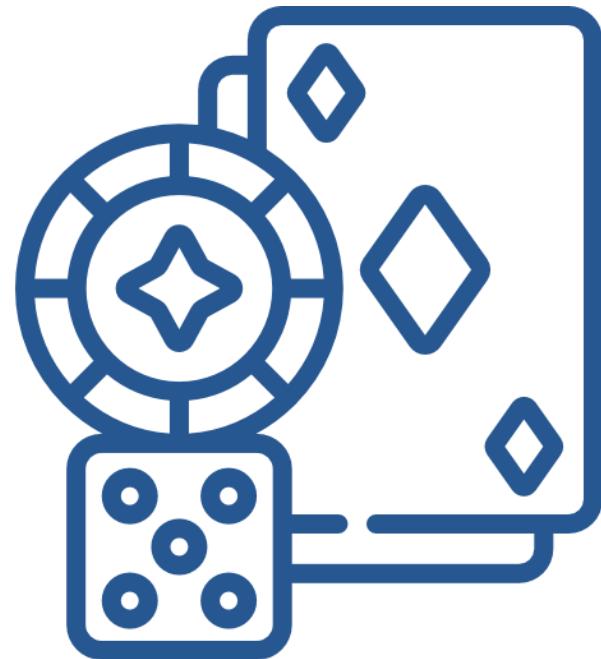
```
number = 123456
for digit n in number:
    print(digit)
```

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] ← List of length 4 created solely to
for i in it_list:           limit the number of iterations
    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")
```

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 integers in a specified range

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

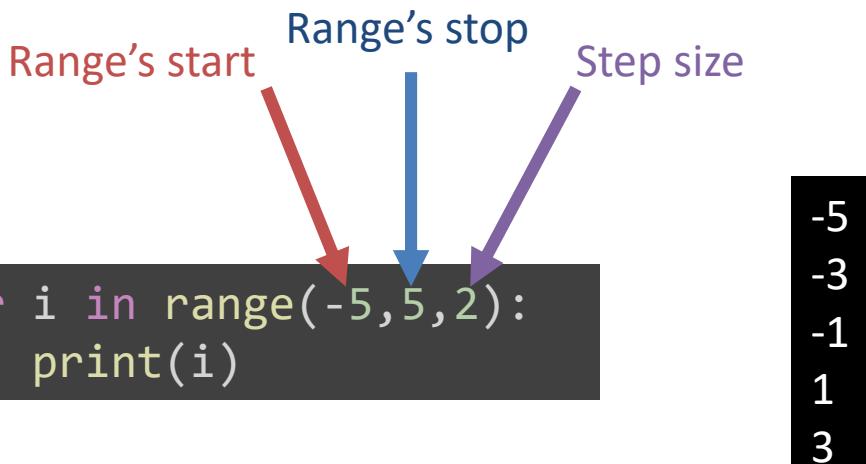
5
6
7
8
9

```
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



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 1 -**
- 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

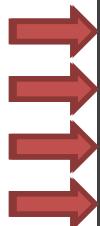
```
t = sum([3, 5, 1])
```

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

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

Generate indexes 0 to $k-1$

Add each number to the total

Return the final 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 iterate 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??
```