

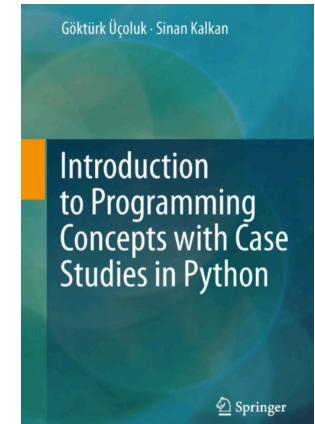
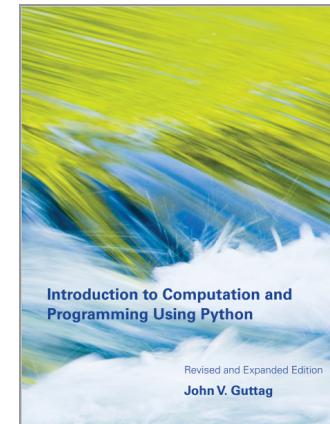
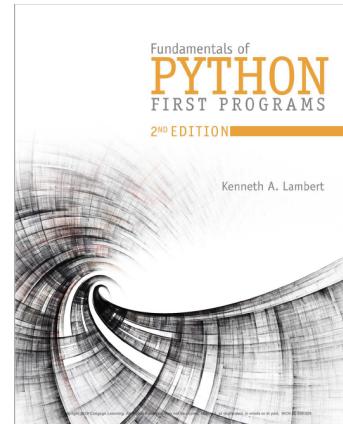
Lecture 1-Introduction

Dr. Hacer Yalım Keleş

hkeles@ankara.edu.tr

Overview

- Introduction to basic concepts
- Programming languages



Disclaimer: The content is prepared using the Chapter 1 of Lambert and Guttag's Books and Chapter 1 of ***Introduction to Programming Concepts with Case Studies in Python*** book.

Algorithms

- Can you describe the steps to add two integers?
- Can you describe the steps to bake a cake?
- Try to write steps:
 - Step 1: Do this
 - Step 2: ...
 - ...
- The sequence of steps that describes the computational processes is called an ALGORITHM.
 - It describes a set of instructions that tell us how to do something, like a recipe.

Sample algorithm to cook custard

- 1- Put custard mixture over heat.
- 2- Stir.
- 3- Dip spoon in custard.
- 4- Remove spoon and run finger across back of spoon
- 5- If the finger path is clear, remove custard from heat and let cool.
- 6- Go to Step 2 and Repeat.



It includes;
some tests for deciding when the process is complete,
instructions about the order in which to execute instructions,
sometimes jumping to some instructions based on a test.

Algorithms in Computer Science

- An algorithm consists of a finite number of instructions.
- Each individual instruction in an algorithm is well defined.
- An algorithm describes a process that eventually halts after arriving at a solution to a problem.
- An algorithm solves a general class of problems.

Information Processing

- In carrying out the instructions of any algorithm, a computing agent manipulates information.
- Information is also referred to as **data** in the modern world of computers.
- Computing agent starts with some given information (**input**), transforms this information with the well defined rules, produces new information (**output**)

Stored-program computers

Creating an instruction-set architecture and detailing the computation as a sequence of instructions, i.e. a program, we make a highly flexible machine.

This machine;

- Stores and manipulates a sequence of instructions
- Has a set of elements that will execute any instruction in that sequence

The heart of a computer becomes a program that can execute any legal set of instructions and thus can be used to compute anything one can describe using some basic set of instructions.

Both data and the program that manipulates data reside in memory.

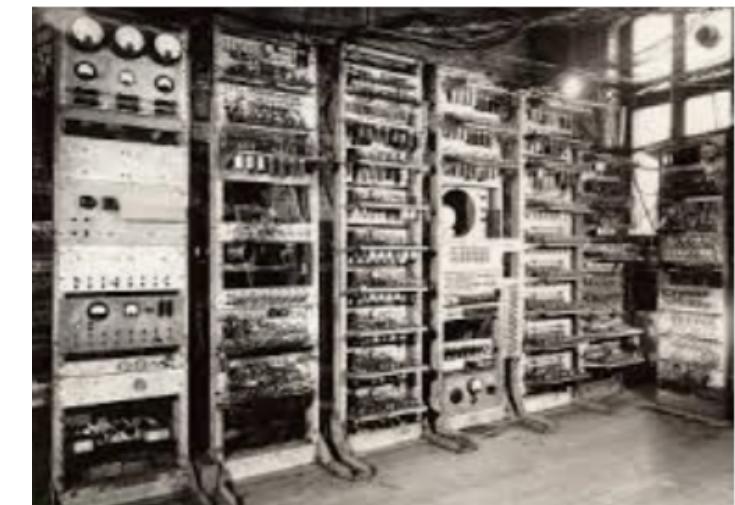
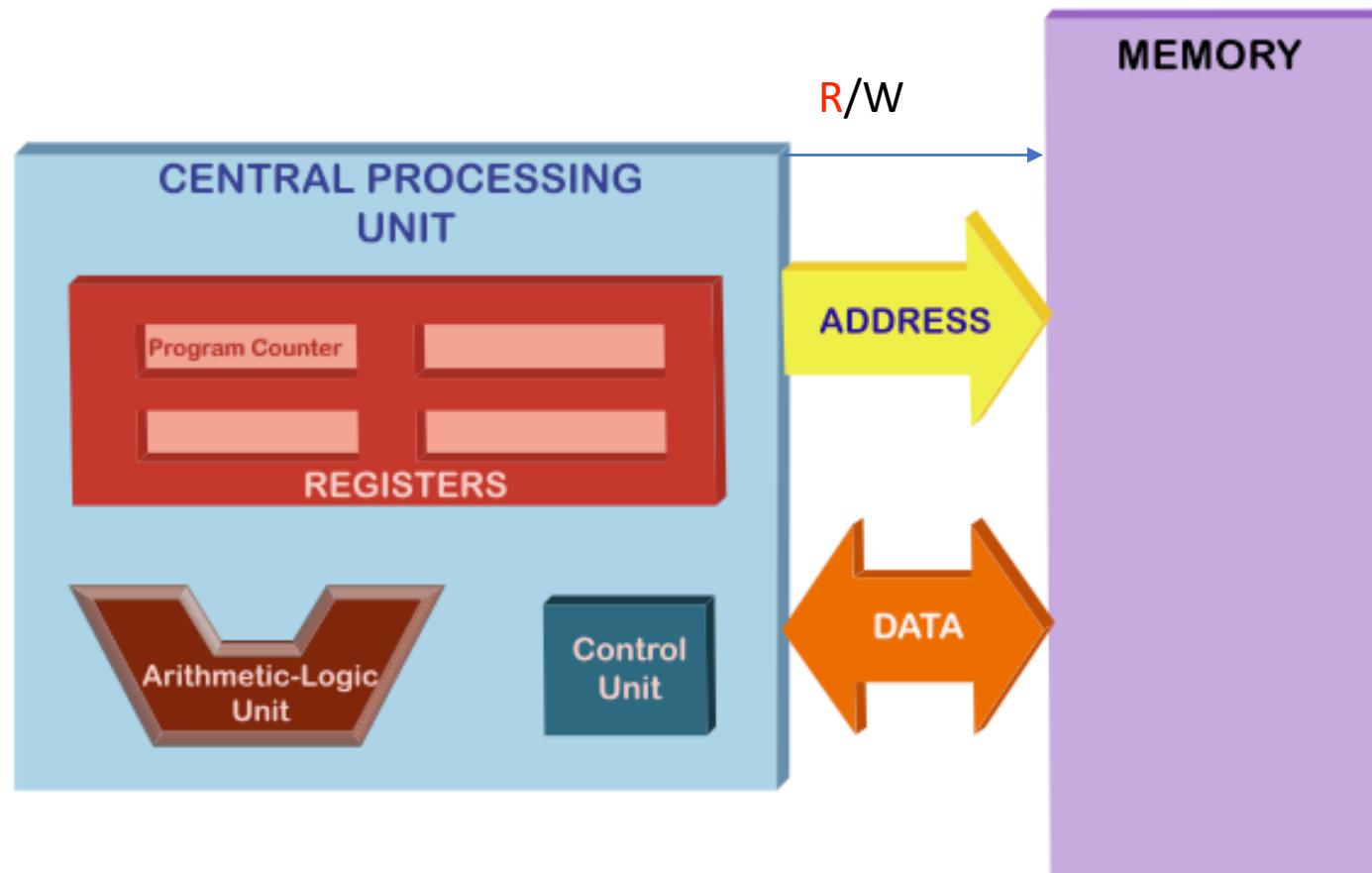


Image is from Wikipedia: Manchester Mark 1 (1949)

Our medium for programming is a Computer.

In particular; “*a Von Neumann Type Computing Machine*”

Von Neumann Architecture



Von Neumann Architecture

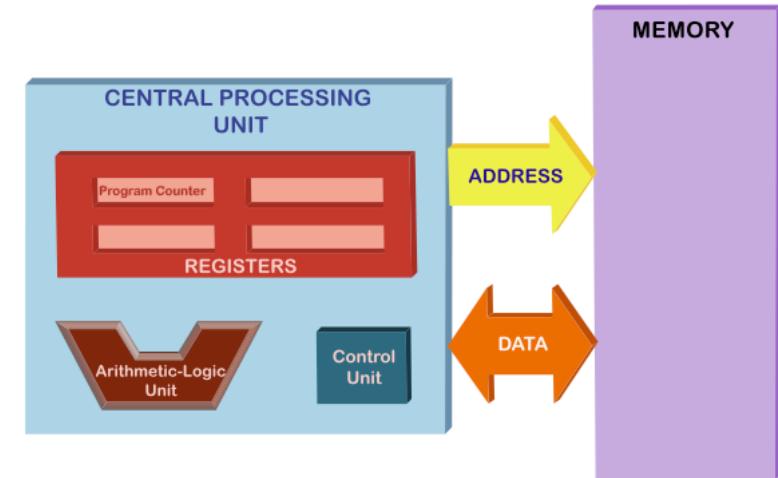
- Digital computing system, based on binary representations

- Consists of CPU & Memory

- CPU perform information (data) processing, such as simple arithmetic and logical operations, accessing to a certain address in memory

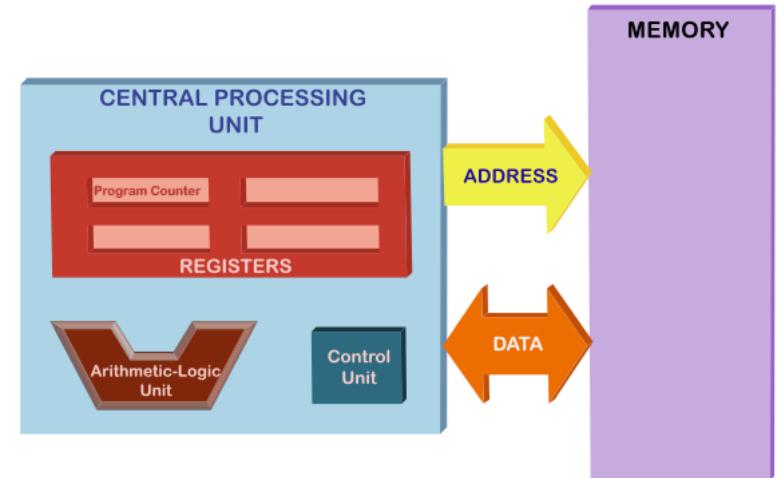
- Memory is a homogenous place that keeps data and instructions. Data is represented in bytes and each byte (i.e. 8 bits) has an address.

- The content in the memory can be overwritten as many times as desired. Once overwritten, it is gone (not recoverable).



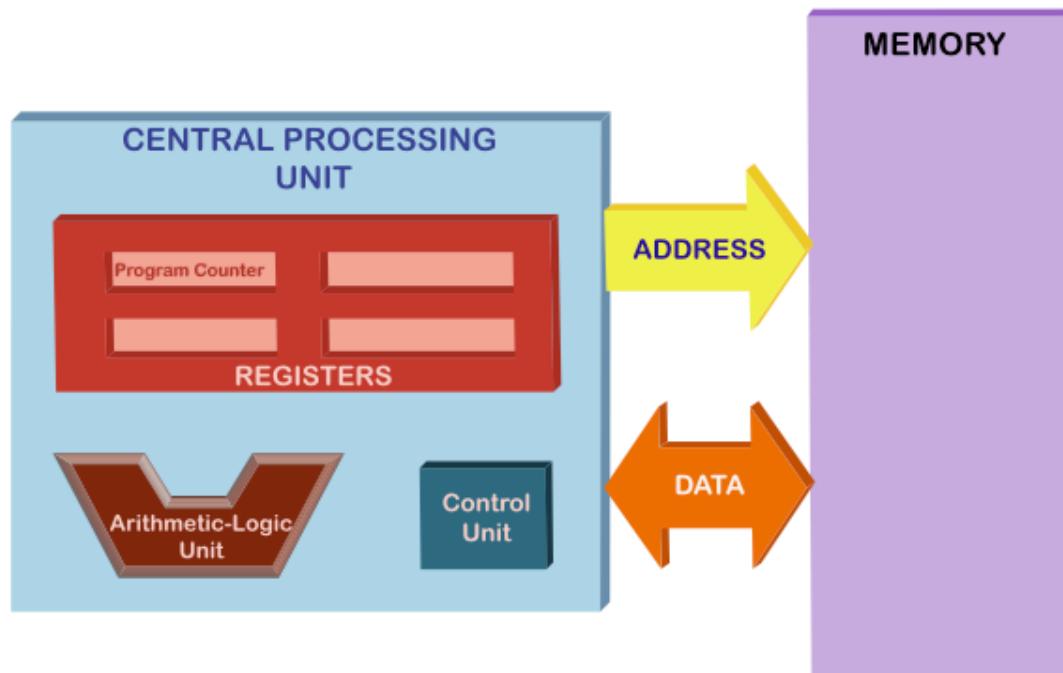
Von Neumann Architecture

-The content in the memory can be overwritten as many times as desired. Once overwritten, it is gone (not recoverable).

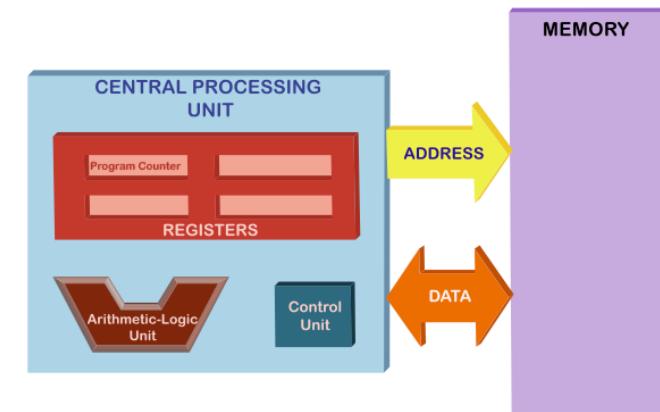


- CPU and Memory communicate through two busses: *address bus* and *data bus*
- There is also another wire from CPU to Memory: *R/W line*

Fetch-Decode-Execute Cycle in VN Architecture



Fetch-Decide-Execute Cycle in VN Architecture



The address bus is unidirectional, but the data bus is bidirectional

CPU sets the address bus to a certain address

CPU sets the R/W line (to read or write)

if the action is W, it sets the data bus the content to write to that address

if the action is R, the Memory loads the data bus reading the content from the memory at the address

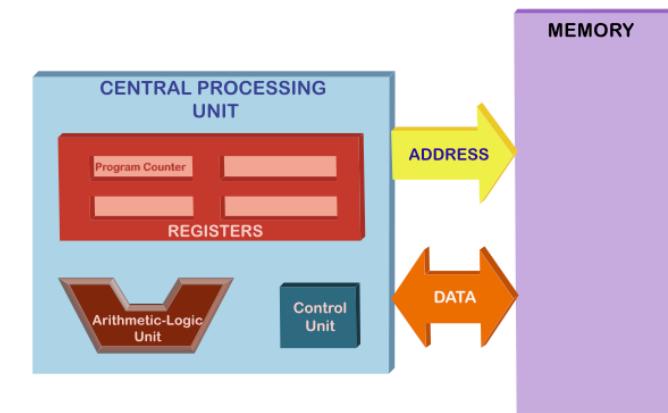


Fetch-Decide-Execute Cycle in VN Architecture

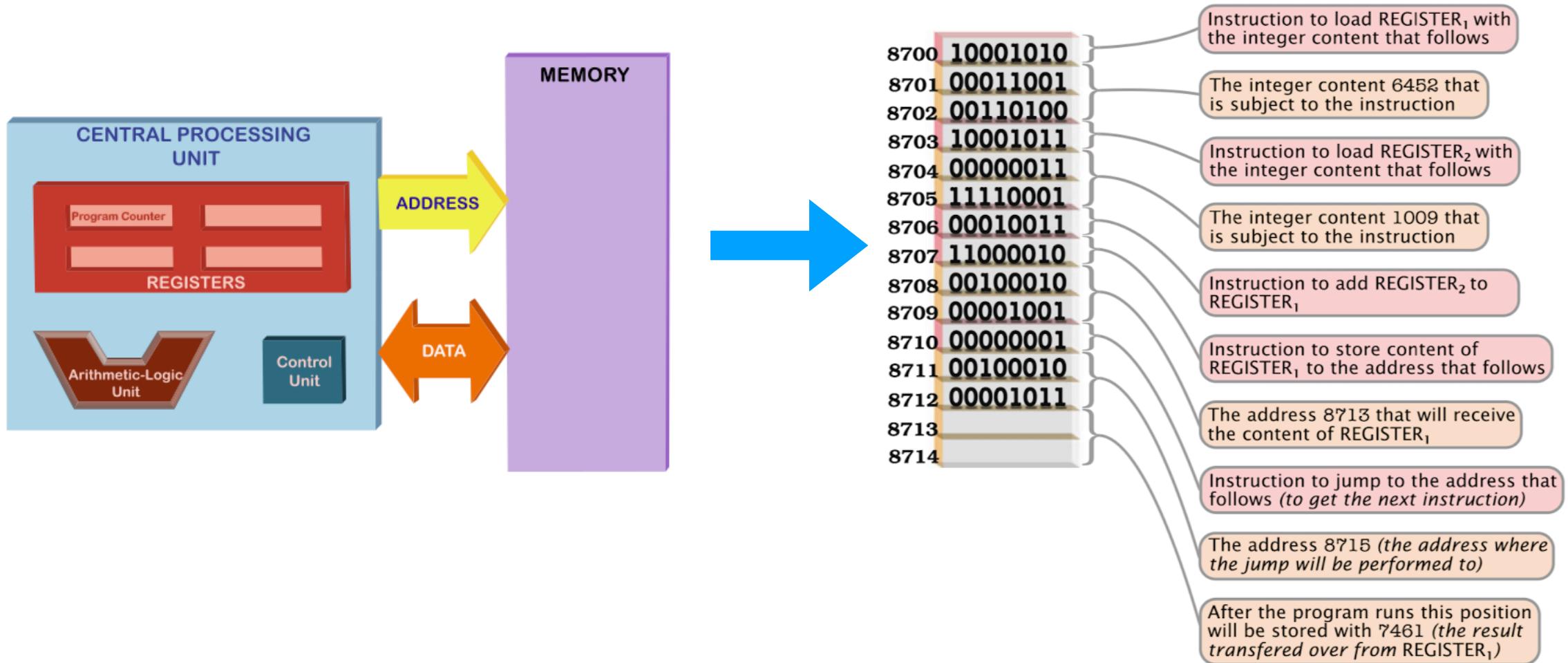
After the CPU performs a unit of action;

- 1- It reads its next action (i.e. instruction): writes the address to the address bus and sets R line
- 2-Memory sends back the instruction over the data bus
- 3- CPU decodes (reads and interprets) the instruction and executes it. (performs whatever is necessary)

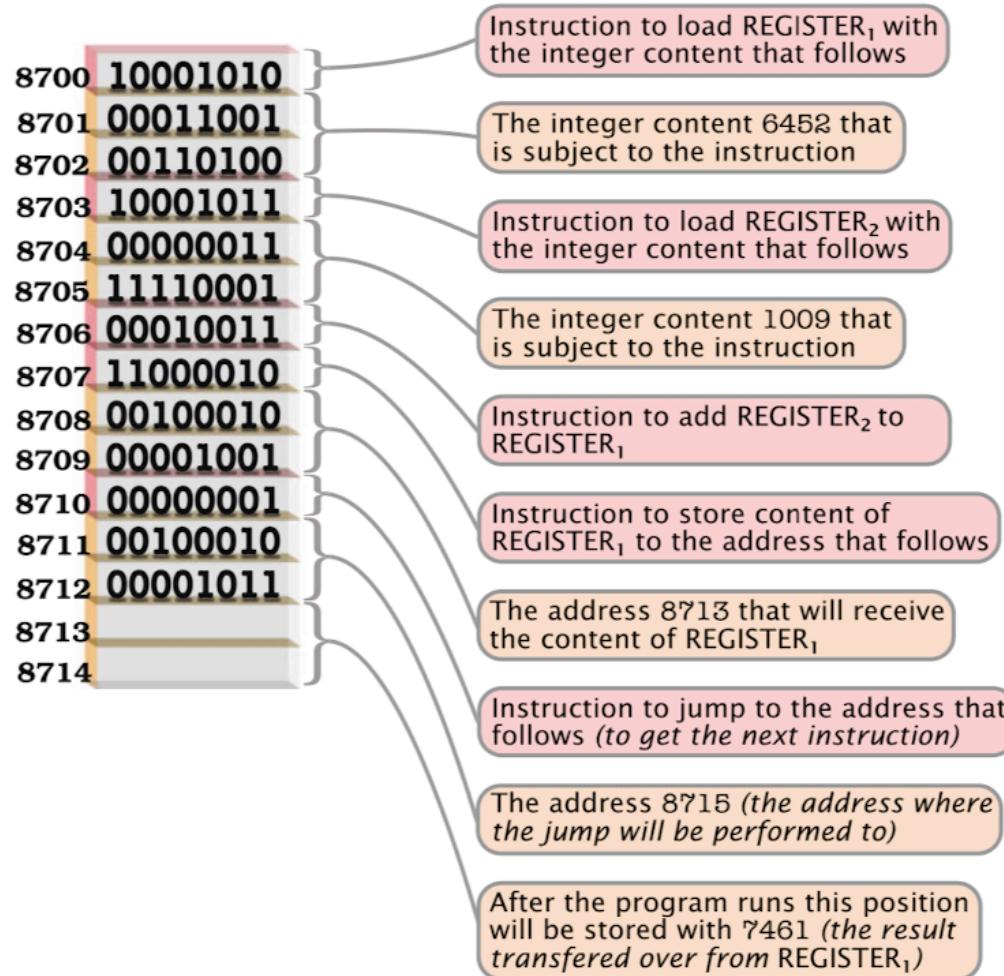
Go fetch the next instruction as in (1)



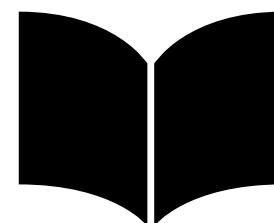
Fetch-Decompile-Execute Cycle in VN Architecture



Memory



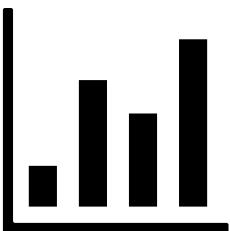
also contains



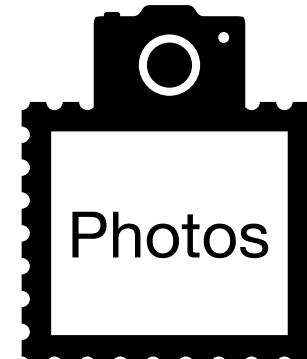
Audio data



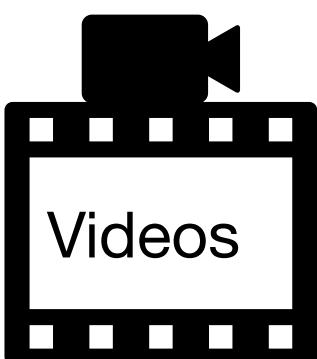
Financial data



Tabular data



Photos



Videos

Technically, programming in Von Neumann Architecture is ...

... generating the correct memory content that will complete the given objective when executed by the CPU.

Overview

The World of Programming

Programming Languages

Programming Paradigms

The Zoo of Programming Languages

How PLs are implemented

How We Write Programs

Meeting with Python

Primary Reference: Chapter 1 of “*Introduction to Programming Concepts with Case Studies in Python*” Book.

Programming Languages

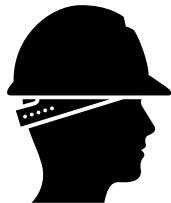
Machine code

- Programming a Von Neumann Architecture manually, byte by byte, is very hard.
- Byte by byte memory layout corresponding to a program is called the *machine code*.

Programming Languages

Low level PLs

A sequence of bytes which multiplies two integer numbers in two different locations in memory
and stores the result in a different location in memory

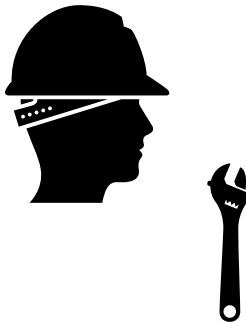


```
01010101 01001000 10001001 11100101 10001011 00010101 10110010 00000011
00100000 00000000 10001011 00000101 10110000 00000011 00100000 00000000
00001111 10101111 11000010 10001001 00000101 10111011 00000011 00100000
00000000 10111000 00000000 00000000 00000000 00000000 11001001 11000011
...
11001000 00000001 00000000 00000000 00000000 00000000
```

Programming Languages

Low level PLs

Assembly; code is a level more in human readable form.
The program does not get shorter, but more readable.



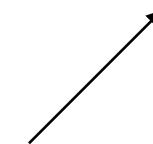
main:

```
pushq  %rbp
movq  %rsp, %rbp
movl  alice(%rip), %edx
movl  bob(%rip), %eax
imull %edx, %eax
movl  %eax, carol(%rip)
```

```
movl  $0, %eax
leave
ret
```

alice:
bob:

```
.long 123
.long 456
```



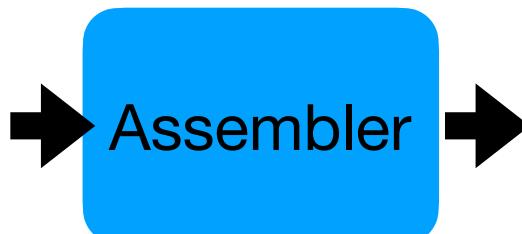
Programming Languages

Low level PLs

Machine code is generated by a program called *assembler*.

main:

```
pushq  %rbp  
movq  %rsp, %rbp  
movl  alice(%rip), %edx  
movl  bob(%rip), %eax  
imull  %edx, %eax  
movl  %eax, carol(%rip)
```



```
01010101 01001000 10001001 11100101 10001011  
00100000 00000000 10001011 00000101 10110000  
00001111 10101111 11000010 10001001 00000101  
00000000 10111000 00000000 00000000 00000000  
...  
11001000 00000001 00000000 00000000 00000000
```

Programming Languages

High level PLs

Why not go one level up (or more)...

If only we had a translator that converts what we write to a machine code for our CPU...



Let us define a language that specifies syntax and semantics of the language

Syntax: the set of rules that define what is allowed to write

Semantics: the meaning of what is written down

Then, we will be free from considering a manual transformation to a specific brand of CPU, its specific set of instructions, register types, counts etc. Cool right? :)

Programming Languages

High level PLs

For over 50 years, around 700 such syntactic and semantic definitions are created...

Most of these languages create high-level concepts:

- human readable form of numbers (decimal, octal, hexadecimal) and strings
- containers to hold data and name them
- expressions similar to the ones we use in mathematics
- constructs for repetitive execution of some parts
- functions that are reusable
- pointers
- facilities for data organization (define new datatypes)

Programming Languages

High level PLs

```
int alice = 123;
int bob = 456;
int carol;
main(void)
{
    carol = alice*bob;
}
```

PLs

Each PL has a set of primitive constructs, a syntax, a static semantics and a semantics.

In English;

the **primitive constructs** are **words**,

the **syntax** describes **which strings of words constitute the well-formed sentences**,

the string: «cat dog boy» is not a syntactically valid sentence

the syntax of English do not support sentences in <noun> <noun> <noun> form

the **static semantics** defines **which sentences are meaningful**,

«I are big» : syntactically valid sentence (<pronoun><verb><adjective>)

it's still not valid, since the noun (I) is singular, and verb (are) is plural. This is a static semantic error.

the **semantics** defines **the meaning of those sentences**.

A language associates a meaning with each syntactically correct string of symbols that has no static semantic errors.

In natural languages the semantics of a sentence can sometimes be ambiguous. However, PLs are designed such that each legal program has exactly one meaning (deterministic)

PLs

In Python;

the **primitive constructs** include literals, e.g. 2.3, 'hello', and arithmetic operators, e.g. +, /.

the **syntax** describes which strings of words constitute the well formed sentences,

3.2 + 6.3 is OK (<literal> <operator> <literal>)

3.2 6.3 is syntactically not OK; the sequences of <literal> <literal> is not supported.

the **static semantics** defines which sentences are meaningful,

3.2/'hello' is syntactically well formed but produces a static semantics error;

it's not meaningful to divide a number by a string

the **semantics** defines the meaning of those sentences.

If a program has no syntactic and static semantics errors, it has a meaning.

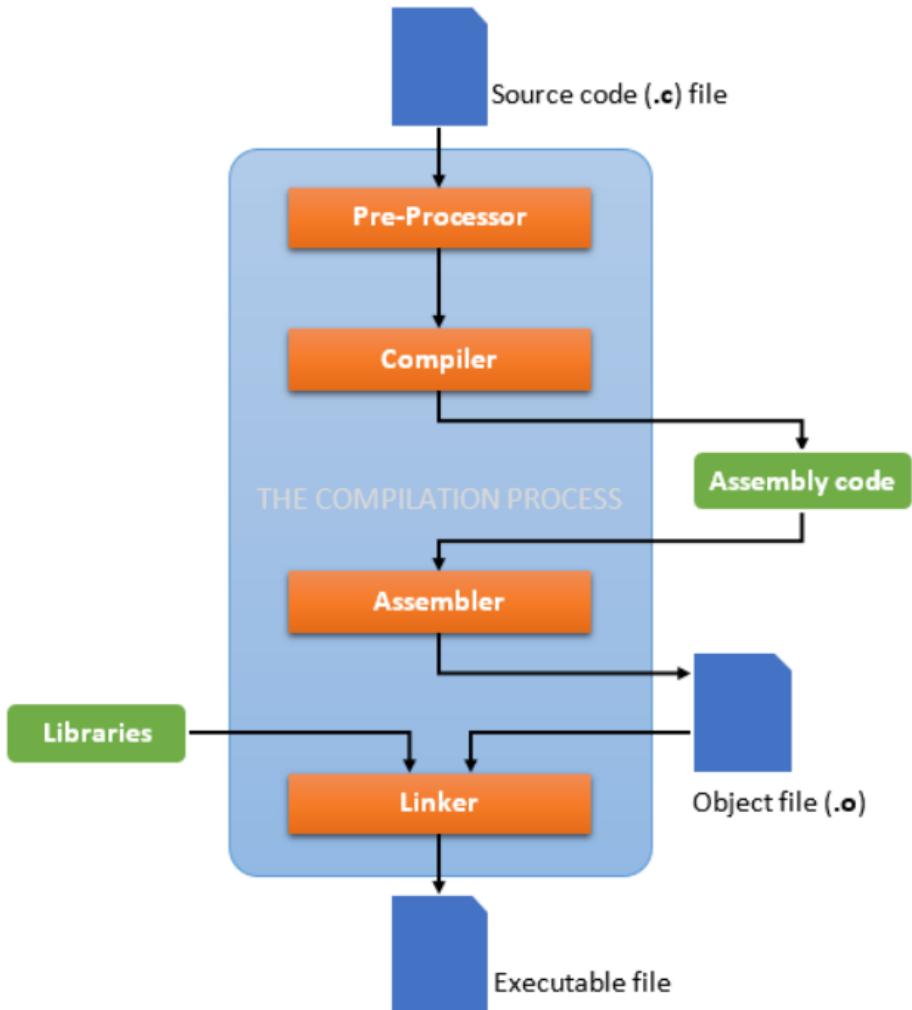
We say «it has semantics».

BUT, it does not mean that all generated programs with semantics has the correct semantic that its creator intended!

What happens when a program runs with wrong semantics?

- It might crash.
It stops running and obviously show that something went wrong.
- It might keep running and running ... without giving any output and no termination.
Is this easy or hard to detect? What do you think?
- It produces a result which is wrong.
Can be fatal;
Imagine you compute a critical parameter for the trajectory of a Space craft to the Mars..
Compute the true doses of radiation therapy for a patient..

Higher level Programming Languages

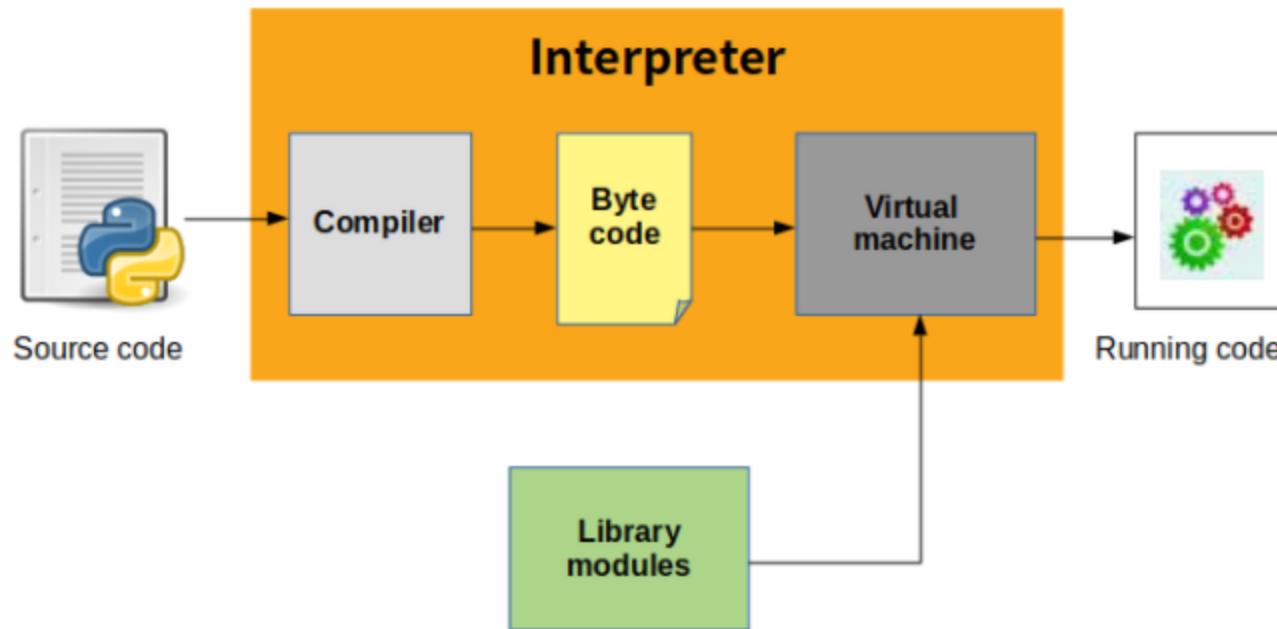


```
class Animal
{
    string m_sName;
public:
    Animal(string name)
        :m_sName(name)
    {}
    virtual const char* Sounds() = 0;
    string GetName(){return m_sName;}
};

class Cat : public Animal
{
    // other necessary member functions for Cat
    // cats Sounds "Miyav"
};
```

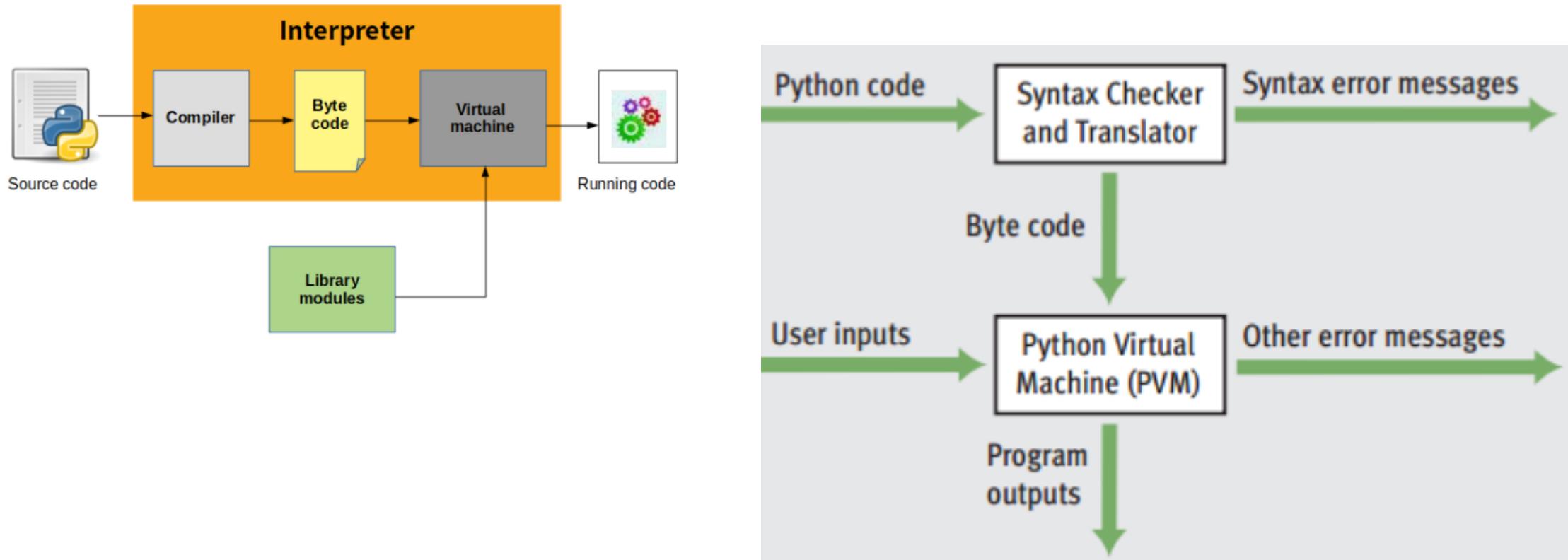
Sample C++ code

Higher level Programming Languages: Python



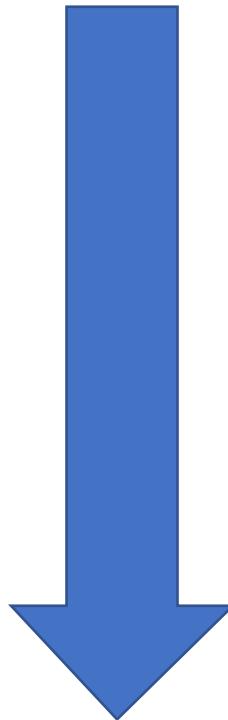
No explicit executable file is generated. Binary code (byte code above) is generated from the source code on the fly and run on a virtual machine.

How Python interpreter works?



Hierarchy of Programming Languages

High-level



- Python, Javascript
 - Interpreted every time it runs
- C, C++
 - Compiled into an executable machine code
- Assembly languages
 - Assembled into machine code
- Machine code
 - Run by CPU

Low-level

Reading Assignment:

- Chapter 1 from Guttag's Book.
- «A Not So-Brief History of Computing Systems»
Chapter 1, from Lambert's Book

What's Next?

- Getting Started Programming with Python!! Yeayy ☺



[Image ref: https://redsignal.net/why-is-programming-fun/](https://redsignal.net/why-is-programming-fun/)

Lecture 2

Introduction to Python

Dr. Hacer Yalım Keleş

hkeles@ankara.edu.tr

Different Aspects of Programming Languages (PLs)

Low-level vs high level:

- using instructions and data objects at the level of the machine
(Assembly languages)
e.g. move 64 bits of data from 0x8416 to 0x9864
- allow using more abstractions (Python, C, C++, Java, etc.)
e.g. object definitions

Different Aspects of Programming Languages (PLs)

General vs targeted to an application domain:

- primitive operations are widely applicable (Python,C,C++,Java, etc.)
- primitive operations are fine-tuned to a domain (Adobe Flash)

Different Aspects of Programming Languages (PLs)

Interpreted vs. compiled:

- source code is executed directly by an interpreter (Python, MATLAB, Ruby)
- source code is first converted to a machine code by a compiler (C, C++)

How about Python?

- A general purpose programming language that can be used effectively to build almost any kind of programs.
- A relatively simple language that is easy to learn
- Provides good runtime feedback to the user, which is very helpful for novice programmers
- There are various freely available libraries that interface Python

It's not optimal for programs that have high reliability constraints due to its weak semantic checking

Python History

- Python was introduced by Guido von Rossum in 1990.
- Python 2.0 was released in 2000.
- Python 3.0 was released in 2008.
 - It's not backward compatible.

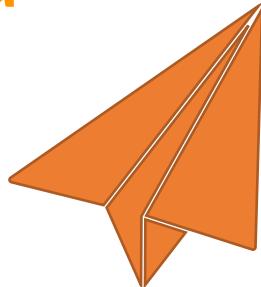
We will use Python 3.6 and above versions in this course.

Our Roadmap

- Basic elements of almost all high-level and general purpose programming languages are conceptually similar.
- To make your PL learning experience easy and fun, we will work with Python 3.x
- We will be focusing on the basic concepts to develop your computational problem solving and algorithmic thinking skills. Python will be just a tool for us.
- We will of course go over the language constructs in necessary detail on the way...

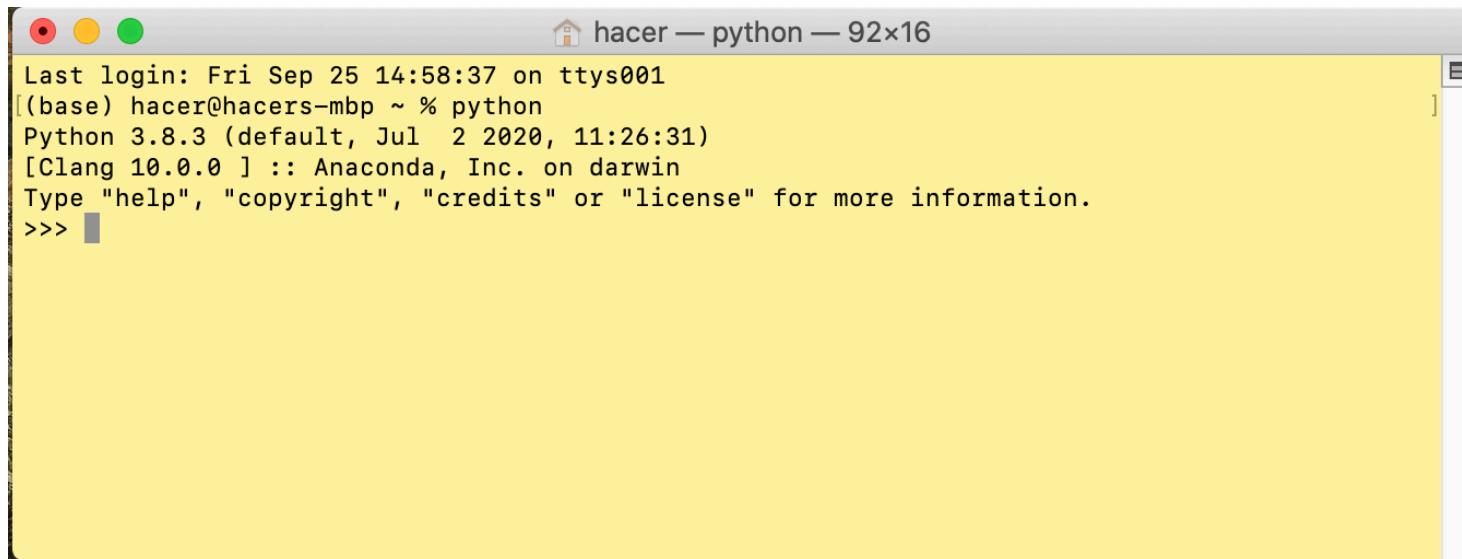
Basic Elements of Python

Fast Forward



Python program

- Also called script: sequences of definitions and commands
- Python commands are executed by Python interpreter in a **shell**,
 - A new shell is created when a new program execution starts
- For the time being, start working in a shell, later we will use some IDEs...A shell looks like this:



The screenshot shows a macOS terminal window with the title bar "hacer — python — 92x16". The window contains the following text:

```
Last login: Fri Sep 25 14:58:37 on ttys001
(base) hacker@hacers-mbp ~ % python
Python 3.8.3 (default, Jul 2 2020, 11:26:31)
[Clang 10.0.0 ] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

Shell

- Write some commands in the shell and see the interpreted results

```
>>> print('Hello Python!')  
Hello Python!
```

- >>> represents the command line
 - I will be using it a lot in assignments etc.
- You can print your Python version:

```
>>> import sys  
>>> print(sys.version)  
3.8.3 (default, Jul 2 2020, 11:26:31)
```

Objects, Expressions, Numerical Types

Objects

- Core to the language, everything in Python 3.x is an object
- Objects have types, which define what we can do with them
- 4 types of **scalar objects**:
 - **int**: -3, 5, 1045, .., 0o25, 0xF4, 0b101
 - **float**: 3.56, .3, 4. , 1.2e-2
 - **bool**: **True**, **False**
 - **None**: **None**.

```
[>>> print(0o26)  
22  
[>>> print(0x26)  
38  
[>>> print(26)  
26  
[>>> print(0b101)  
5
```

```
[>>> print(3.6)  
3.6  
[>>> print(.4)  
0.4  
[>>> print(2.)  
2.0
```



```
[>>> print(2.3e-4)  
0.00023  
[>>> print(1.2e+3)  
1200.0  
[>>> print(3.4e2)  
340.0
```

Operators

- Objects and operators can be combined to form **expressions**.
 - Each expression evaluates to an object of some type, which has a **value**.
 - $3+2 \rightarrow$ the object **5** of type **int**
 - $3.2 + 4.0 \rightarrow$ the object **7.2** of type **float**
 - $2 == 4 \rightarrow$ the object **False** of type **bool**

```
[>>> type(3+2)
<class 'int'>
[>>> type(3.0+2.0)
<class 'float'>
```

```
[>>> type(2==3)
<class 'bool'>
[>>> 2==3
False
```

Operators on type int and float

i+j is the sum of i and j. If i and j are both of type int, the result is an int. If either of them is a float, the result is a float.

i-j is i minus j. If i and j are both of type int, the result is an int. If either of them is a float, the result is a float.

i*j is the product of i and j. If i and j are both of type int, the result is an int. If either of them is a float, the result is a float.

i//j is integer division. For example, the value of $6//2$ is the int 3 and the value of $6//4$ is the int 1. The value is 1 because integer division returns the quotient and ignores the remainder.

i/j is i divided by j. In Python 2.7, when i and j are both of type int, the result is also an int, otherwise the result is a float. In this book, we will never use / to divide one int by another. We will use // to do that. (In Python 3, the / operator, thank goodness, always returns a float. For example, in Python 3 the value of $6/4$ is 1.5.)

i%j is the remainder when the int i is divided by the int j. It is typically pronounced “i mod j,” which is short for “i modulo j.”

ij** is i raised to the power j. If i and j are both of type int, the result is an int. If either of them is a float, the result is a float.

The comparison operators are == (equal), != (not equal), > (greater), >= (at least), < (less) and <= (at most).

Operators on type bool

a and b is True if both a and b are True, and False otherwise.

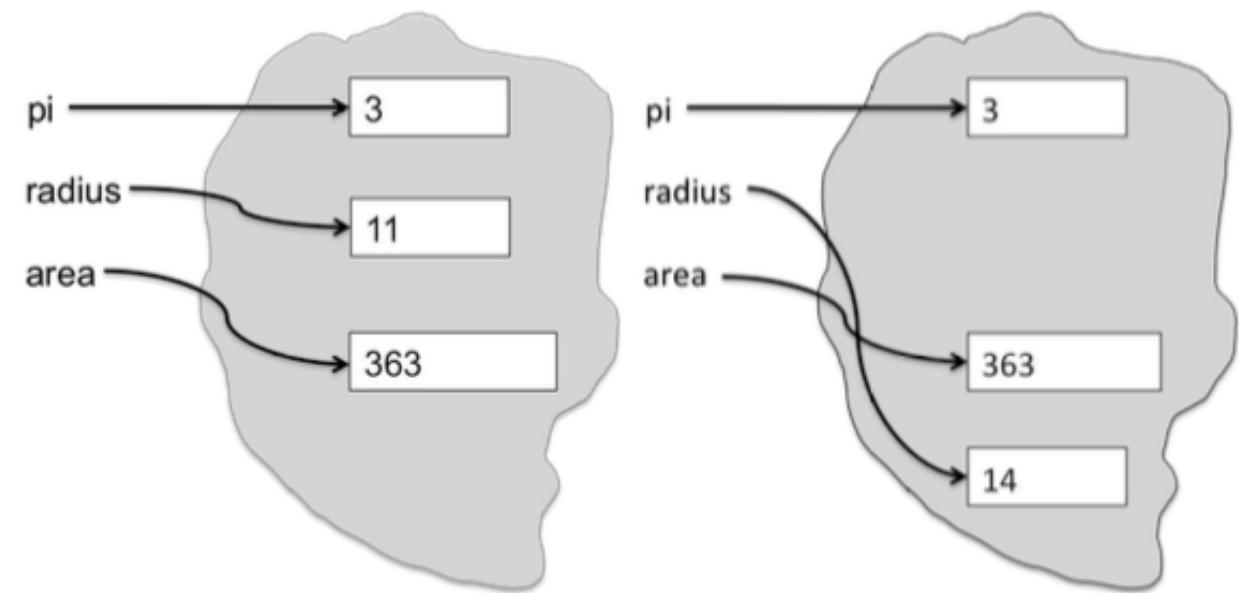
a or b is True if at least one of a or b is True, and False otherwise.

not a is True if a is False, and False if a is True.

Variables and Assignment

- Variables provide a way to **associate names with objects**:

- `pi = 3`
- `radius = 11`
- `area = pi * (radius**2)`
- `radius = 14`



Note that;

- In Python, **a variable is just a name, nothing more.**
- An assignment associates the name to the left of the = symbol with the object to the right of the = symbol.
- An object can have one, more than one or no name associated with it.

Variables

- Variable names can contain uppercase or lowercase letters and digits and _; **they can not start with a digit.**
- Case matters: radius and Radius are different variables, **be careful!**
- Reserved keywords are not used as variable names:

and	except	lambda	with
as	finally	nonlocal	while
assert	false	None	yield
break	for	not	
class	from	or	
continue	global	pass	
def	if	raise	
del	import	return	
elif	in	True	
else	is	try	

Reserved words for Python 3

Comments

#

- In a line, text following the symbol # are not interpreted by Python

Comment

```
#subtract area of square s from area of circle c
areaC = pi*radius**2
areaS = side*side
difference = areaC-areaS
```

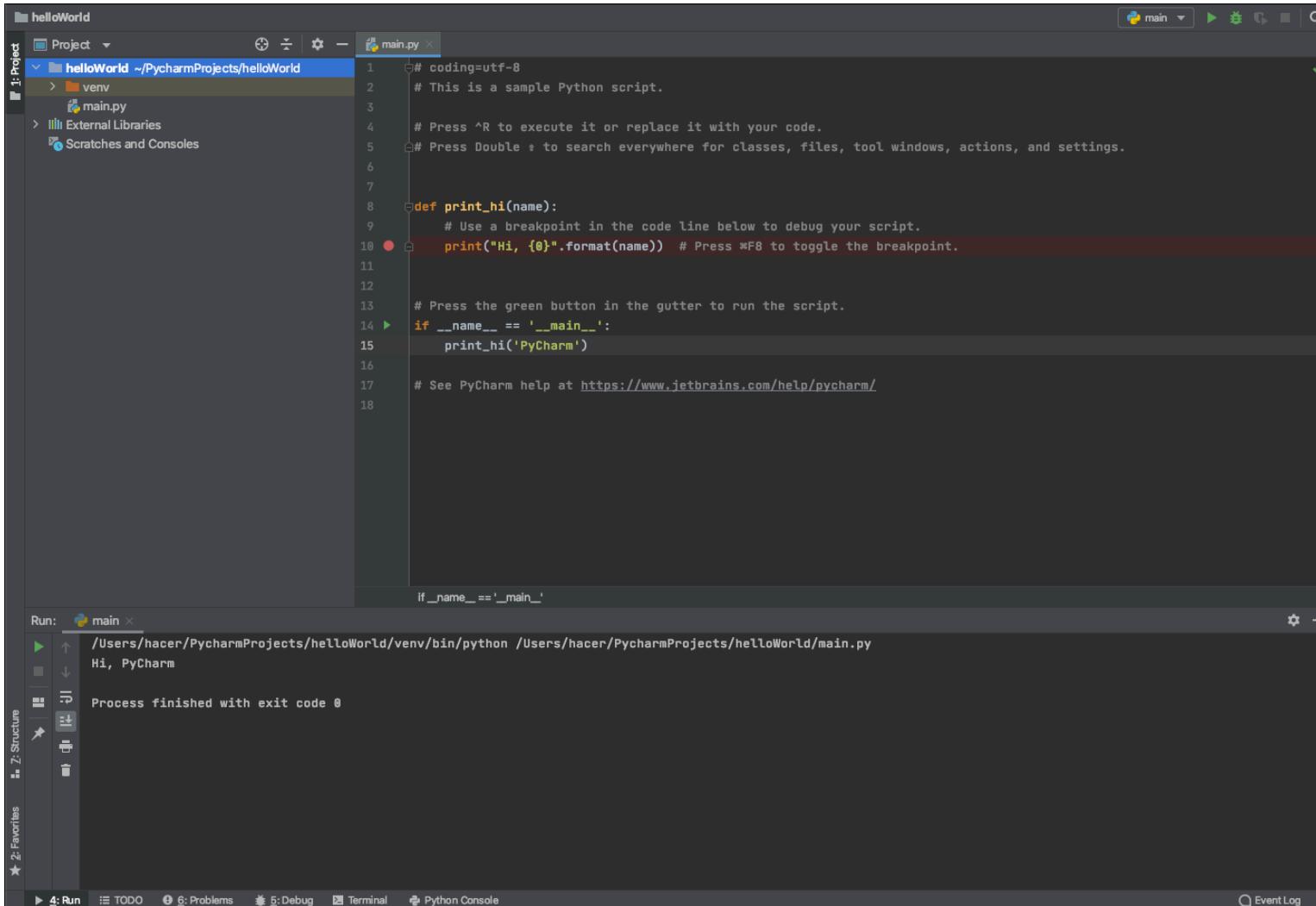
Multiple assignment

- Python allows multiple assignment
 - E.g. `x, y = 2, 3` -> binds `x` to 2, `y` to 3
- All of the expressions on the RHS of the assignments are evaluated before bindings are changed.

```
>>> x, y = 2, 3
>>> x, y = y, x
>>> print('x = ', x)
x =  3
>>> print('y = ', y)
y =  2
```

Integrated Development Environments

IDE



The screenshot shows the PyCharm IDE interface. The top navigation bar includes tabs for 'main' (selected), 'Run', 'TODO', 'Problems', 'Debug', 'Terminal', and 'Python Console'. The bottom status bar shows 'Event Log'.

The main window displays a Python project named 'helloWorld' located at '~/PycharmProjects/helloWorld'. The project structure includes a 'venv' folder and a 'main.py' file. The 'main.py' code is as follows:

```
# coding=utf-8
# This is a sample Python script.

# Press ^R to execute it or replace it with your code.
# Press Double ⇧ to search everywhere for classes, files, tool windows, actions, and settings.

def print_hi(name):
    # Use a breakpoint in the code line below to debug your script.
    print("Hi, {0}".format(name)) # Press ⌘F8 to toggle the breakpoint.

if __name__ == '__main__':
    print_hi('PyCharm')

# See PyCharm help at https://www.jetbrains.com/help/pycharm/
```

A red dot indicates a breakpoint on the first line of the 'print_hi' function. The 'Run' tool window at the bottom shows the command: '/Users/hacer/PycharmProjects/helloWorld/venv/bin/python /Users/hacer/PycharmProjects/helloWorld/main.py'. The output pane shows the result: 'Hi, PyCharm' and 'Process finished with exit code 0'.

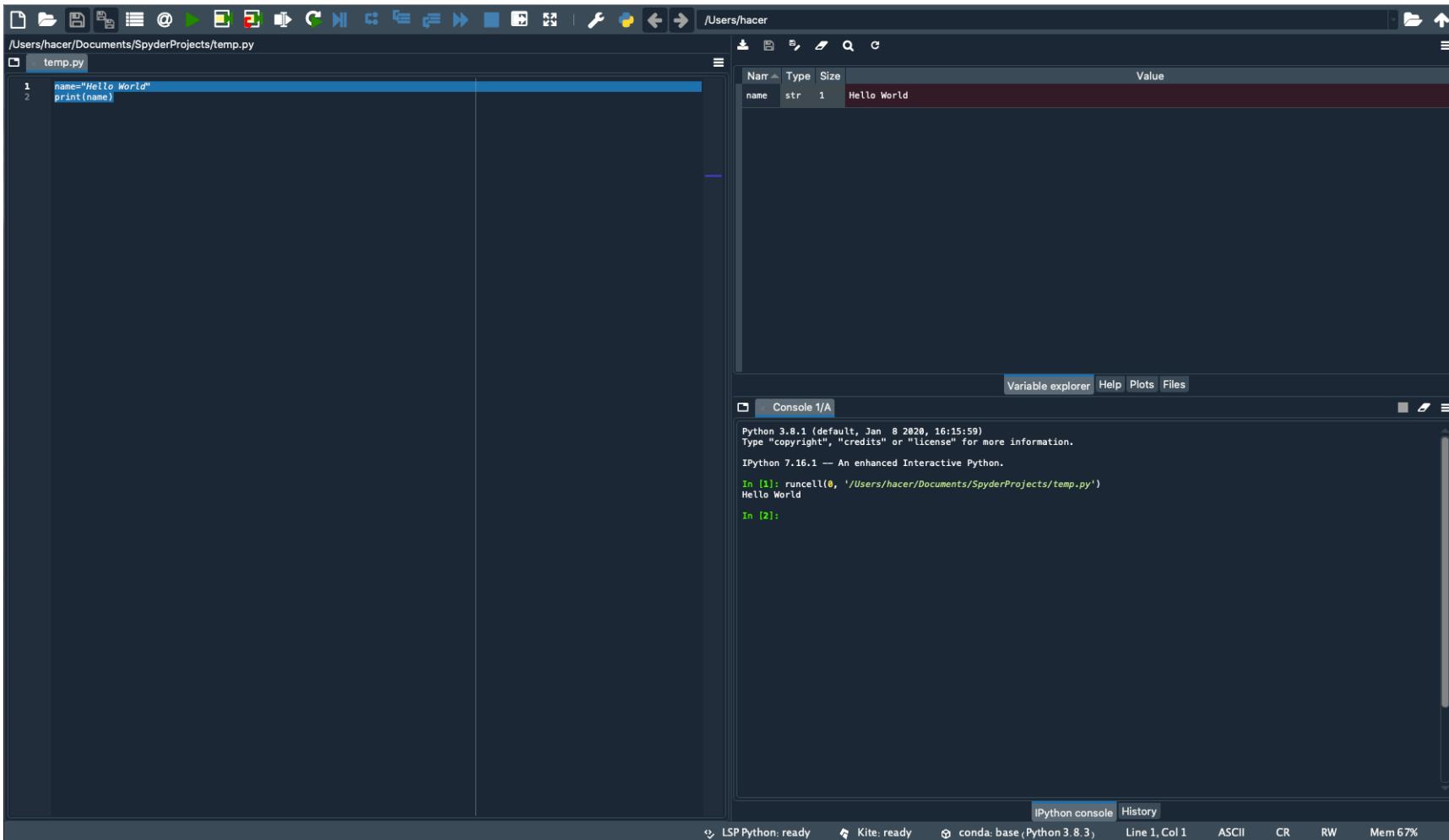
PyCharm

Integrated Development Environments

IDE



Spyder

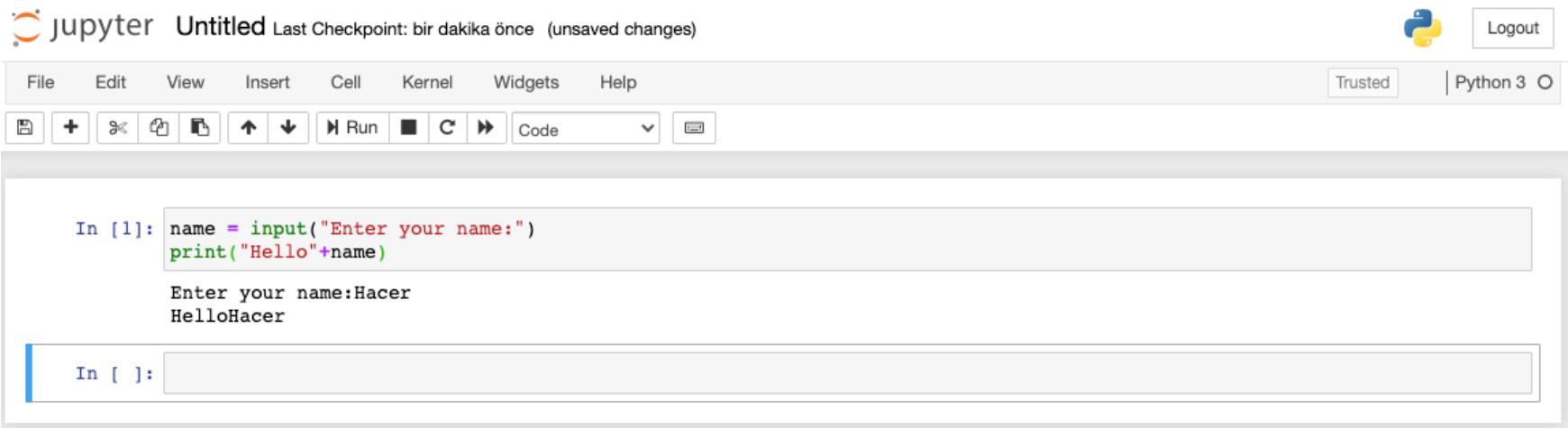


Spyder - Comes with Anaconda installation

ANACONDA NAVIGATOR

Integrated Development Environments

IDE



```
Last login: Sun Oct 11 18:04:55 on ttys000
/Users/hacer/opt/anaconda3/bin/jupyter_mac.command ; exit;
(base) haker@hakers-mbp ~ % /Users/hacer/opt/anaconda3/bin/jupyter_mac.command ; exit;
[I 18:14:56.229 NotebookApp] Writing notebook server cookie secret to /Users/hacer/Library/Jupyter/runtime/notebook_cookie_secret
[I 18:14:56.593 NotebookApp] JupyterLab extension loaded from /Users/hacer/opt/anaconda3/lib/python3.8/site-packages/jupyterlab
[I 18:14:56.593 NotebookApp] JupyterLab application directory is /Users/hacer/opt/anaconda3/share/jupyter/lab
[I 18:14:56.596 NotebookApp] Serving notebooks from local directory: /Users/hacer
[I 18:14:56.596 NotebookApp] The Jupyter Notebook is running at:
[I 18:14:56.596 NotebookApp] http://localhost:8888/?token=d59bf430ce513098ed1c57c0b85f038dd1f8ed1dee2ab24d
[I 18:14:56.596 NotebookApp] or http://127.0.0.1:8888/?token=d59bf430ce513098ed1c57c0b85f038dd1f8ed1dee2ab24d
[I 18:14:56.596 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip configuration).
[C 18:14:56.603 NotebookApp]

To access the notebook, open this file in a browser:
file:///Users/hacer/Library/Jupyter/runtime/nbserver-58732-open.html
Or copy and paste one of these URLs:
http://localhost:8888/?token=d59bf430ce513098ed1c57c0b85f038dd1f8ed1dee2ab24d
or http://127.0.0.1:8888/?token=d59bf430ce513098ed1c57c0b85f038dd1f8ed1dee2ab24d
[I 18:17:32.654 NotebookApp] Creating new notebook in /Documents/SpyderProjects
[I 18:17:32.659 NotebookApp] Writing notebook-signing key to /Users/hacer/Library/Jupyter/notebook_secret
[I 18:17:33.313 NotebookApp] Kernel started: f9a53f9e-6edb-4a70-8863-2c8a34cdde4d
[I 18:19:34.036 NotebookApp] Saving file at /Documents/SpyderProjects/Untitled.ipynb
```



Jupiter Notebook - Comes with Anaconda installation

ANACONDA NAVIGATOR

Lecture 3

Python Basics

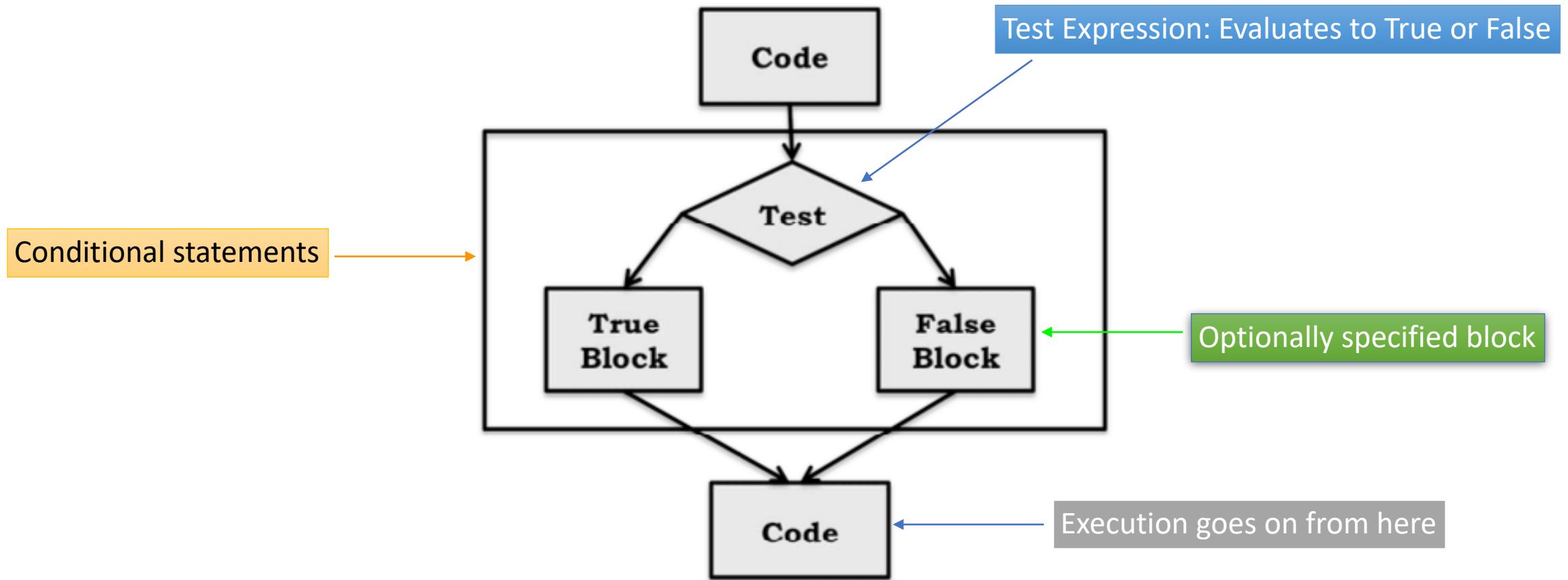
Conditional Statements

Dr. Hacer Yalım Keleş

hkeles@ankara.edu.tr

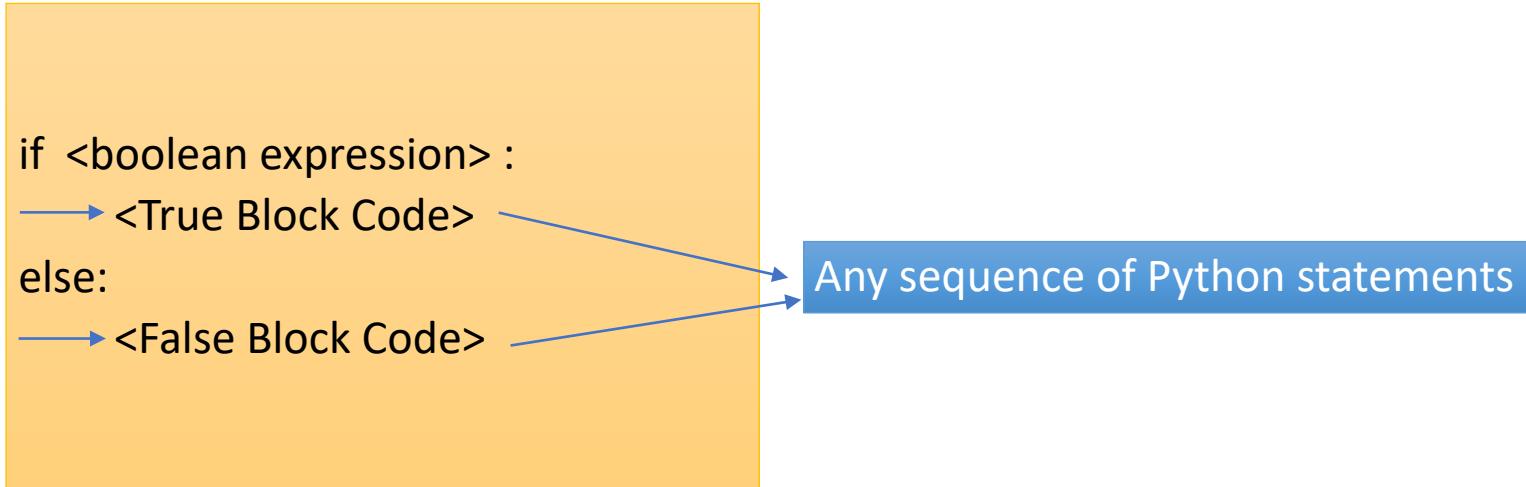
Disclaimer: Content of this lecture slides are mostly from **Chapter 2** of Guttag's Book: "*Introduction to Computation and Programming Using Python*".

Branching: Conditional Statements



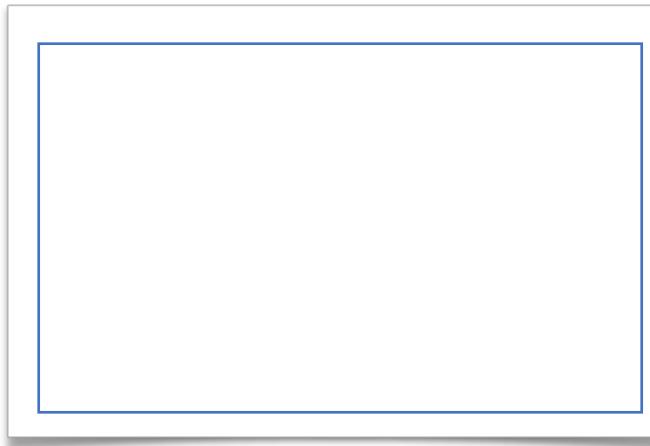
Branching

Indentation issue
Indentation issue



Example:

Write a program that prints Even if a given number is even, Odd otherwise.



Example:

Write a program that prints Even if a given number is even, Odd otherwise.

```
>>> number = 12
>>> if number%2 == 0:
...     print("Even")
... else:
...     print("Odd")
...
Even
```

Example:

Write a program that prints Even if a given number is even, Odd otherwise.

Indentation!
Indentation!

Result

```
>>> number = 12
>>> if number%2 == 0:
...     print("Even")
... else:
...     print("Odd")
...
Even
```

Nested branching

You can include additional branches in the True code block or False code block. This is called Nested conditional statements

```
if x%2 == 0:  
    if x%3 == 0:  
        print("divisible by 2 and 3")  
    else:  
        print("divisible by 2 and not 3")  
elif x%3 == 0:  
    print("divisible by 3 and not 2")
```

Example:

Write a program that finds the smallest of 3 variables x, y, z.

Let us write this using Logical Expressions:

Example:

Write a program that finds the smallest of 3 variables x, y, z.

Let us write this using Logical Expressions:

```
if x<y and x<z:  
    print("x")  
elif y<z:  
    print("y")  
else:  
    print("z")
```

Query:

How much time does execution of these programs take?

Assume that each line of code takes one unit of time...

```
a = 3  
b = 5  
add = a + b  
multi = a*b  
print(add)  
print(multi)
```

```
if x<y and x<z:  
    print("x")  
elif y<z:  
    print("y")  
else:  
    print("z")
```

Query:

How much time does execution of these programs take?

Assume that each line of code takes one unit of time...

```
a = 3  
b = 5  
add = a + b  
multi = a*b  
print(add)  
print(multi)
```

6

```
if x<y and x<z:  
    print("x")  
elif y<z:  
    print("y")  
else:  
    print("z")
```

<=6

Computational Complexity

A program for which the maximum running time is bounded by the length of the program is said to run in **constant time**. This does not mean that each time it is run it executes the same number of steps. It means that there exists a constant, k , such that the program is guaranteed to take no more than k steps to run. This implies that the running time does not grow with the size of the input to the program.

Assignment:

Write a program that examines three variables— x , y , and z — and prints the largest odd number among them. If none of them are odd, it should print a message to that effect.

TODO AT HOME...

Lecture 5

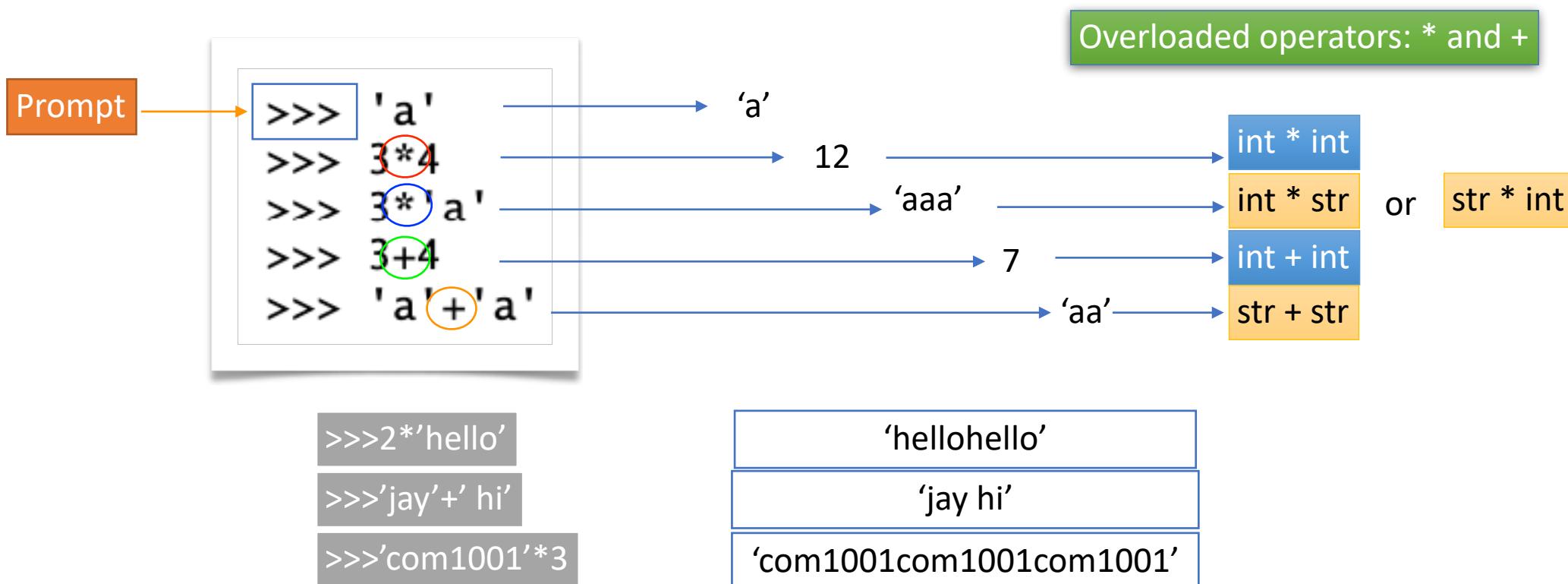
Iteration

Dr. Hacer Yalım Keleş

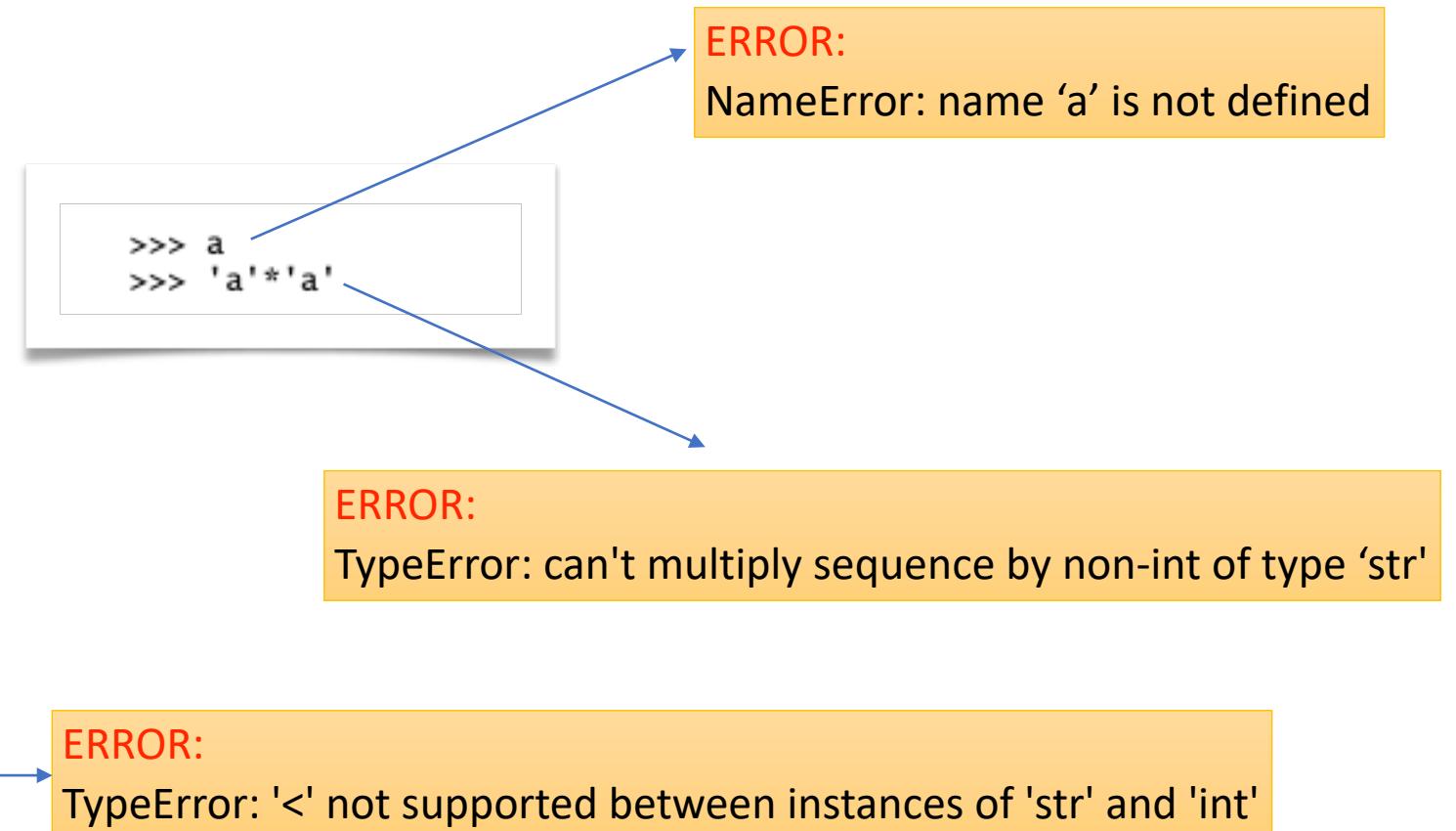
hkeles@ankara.edu.tr

Disclaimer: Content of this lecture slides are mostly from **Chapter 2** of Guttag's Book: "*Introduction to Computation and Programming Using Python*".

Try these:



Type checking



Useful string functions

Length of a string: `>>>len('happy')` 5

Indexing:

<code>>>>'happy'[0]</code>	'h'
<code>>>>'happy'[1]</code>	'a'
<code>>>>'happy'[4]</code>	'y'
<code>>>>'happy'[5]</code>	<code>IndexError: string index out of range</code>

Reverse Indexing:

<code>>>>'happy'[-1]</code>	'y'
<code>>>>'happy'[-3]</code>	'p'
<code>>>>'happy'[-5]</code>	'h'
<code>>>>'happy'[-6]</code>	<code>IndexError: string index out of range</code>

Slicing:

<code>>>>'happy'[1:3]</code>	'ap'
<code>>>>'happy'[2:5]</code>	'ppy'
<code>>>>'happy'[0:7]</code>	'happy'
<code>>>>'happy'[0:3]</code>	'hap'

<code>>>>'happy'[0:len('abc')]</code>	
<code>>>>'happy'[:3]</code>	'hap'
<code>>>>'happy'[1:]</code>	'appy'
<code>>>>'happy'[:]</code>	'happy'

'hap' `>>>'happy'[0:-1]` 'happ'

<code>>>>'happy'[0:-2]</code>	'hap'
<code>>>>'happy'[1:-2]</code>	'ap'
<code>>>>'happy'[3:-4]</code>	<code>''</code> Empty str

Input

>>>raw_input

Not defined in Python 3 anymore!!

>>>input

returns input as str

```
>>> name = input('What is your name? ')
What is your name? Hacer
>>> age = input('Hi '+name+'!\nHow old are you? ')
Hi Hacer!
How old are you? 41
```

Input

>>>raw_input

Not defined in Python 3 anymore!!

>>>input

returns input as str

```
>>> name = input('What is your name? ')
What is your name? Hacer
>>> age = input('Hi '+name+'!\nHow old are you? ')
Hi Hacer!
How old are you? 41
>>> print('No way! You must be kidding :)\n', 'So, you were born in ', 2020-age, ', right?')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>    What's happening??
TypeError: unsupported operand type(s) for -: 'int' and 'str'
```

Input

>>>raw_input

Not defined in Python 3 anymore!!

>>>input

returns input as str

```
>>> name = input('What is your name? ')
What is your name? Hacer
>>> age = input('Hi '+name+'!\nHow old are you? ')
Hi Hacer!
How old are you? 41
>>> print('No way! You must be kidding :)\n','So, you were born in ',2020-age,' right?')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'int' and 'str'
```



age is a str

Input

>>>raw_input → Not defined in Python 3 anymore!!

>>>input → returns input as str

```
[>>> name = input('What is your name? ')
What is your name? Hacer
[>>> age = input('Hi '+name+'!\nHow old are you? ')
Hi Hacer!
How old are you? 41
[>>> print('No way! You must be kidding :)\n','So, you were born in ',2020-age, ', 2020-age, right?')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'int' and 'str'
[>>> print('No way! You must be kidding :)\n','So, you were born in ',2020-int(age), ', right?')
No way! You must be kidding :
So, you were born in 1979 , right?
```



Typecast age to int

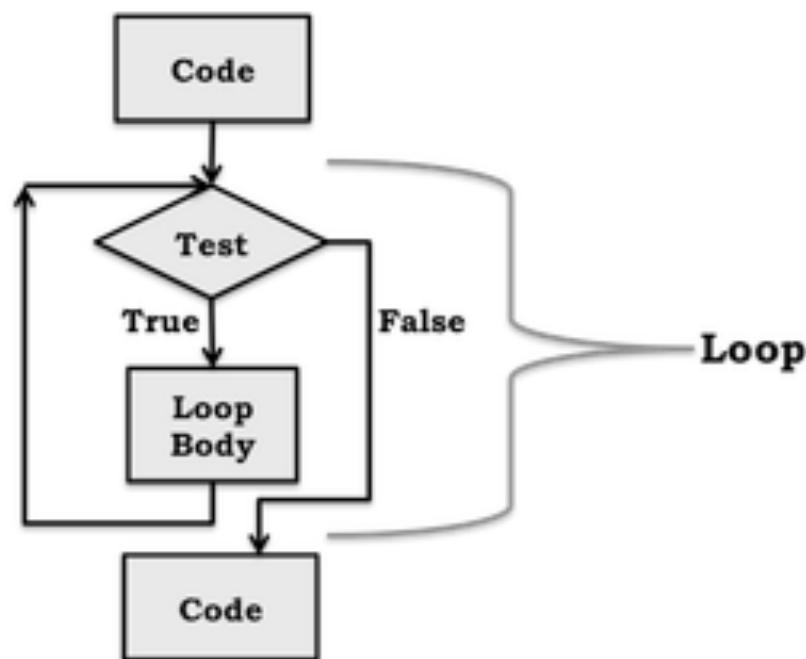
Type Casting

Type conversions (also called **type casts**) are used often in Python code. We use the name of a type to convert values to that type. So, for example, the value of `int('3')*4` is 12. When a float is converted to an int, the number is truncated (not rounded), e.g., the value of `int(3.9)` is the int 3.

Iteration

while loop

Flow chart for iteration



Compute the square of an integer

```
x=4
ans = 0
itersLeft = x
while (itersLeft != 0):
    ans = ans + x
    itersLeft = itersLeft - 1
print(str(x) + '*' + str(x) + ' = ' + str(ans))
```

Compute the square of an integer

```
x=4
ans = 0
itersLeft = x
while (itersLeft != 0):
    print('ans: ' + str(ans) + 'itersLeft: ' + str(itersLeft)) #tracing variables
    ans = ans + x
    itersLeft = itersLeft - 1

print('ans: ' + str(ans) + 'itersLeft: ' + str(itersLeft)) #the values at the end
print('\n\n')
print(str(x) + '*' + str(x) + ' = ' + str(ans))
```

Compute the square of an integer

```
x=4
ans = 0
itersLeft = x
while (itersLeft != 0):
    print('ans: ' + str(ans) + 'itersLeft: ' + str(itersLeft)) #tracing variables
    ans = ans + x
    itersLeft = itersLeft - 1

print('ans: ' + str(ans) + 'itersLeft: ' + str(itersLeft)) #the values at the end
print('\n\n')
print(str(x) + '*' + str(x) + ' = ' + str(ans))
```

```
ans: 0itersLeft: 4
ans: 4itersLeft: 3
ans: 8itersLeft: 2
ans: 12itersLeft: 1
ans: 16itersLeft: 0
```

```
4*4 = 16
```

Compute the square of an integer

For what values of x ,
does this algorithm
work?

```
x=4
ans = 0
itersLeft = x
while (itersLeft != 0):
    ans = ans + x
    itersLeft = itersLeft - 1
print(str(x) + '*' + str(x) + ' = ' + str(ans))
```

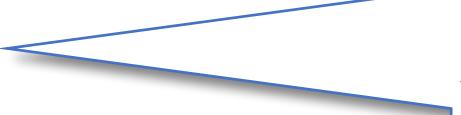
Compute the square of an integer

Please rewrite this algorithm so that it works for any number

TODO on your own at home..

```
x=4
ans = 0
itersLeft = x
while (itersLeft != 0):
    ans = ans + x
    itersLeft = itersLeft - 1
print(str(x) + '*' + str(x) + ' = ' + str(ans))
```

Try this:



Write a code that takes a positive integer N from the input and returns the sum of the numbers from 1 to N , N included.

Write a code that takes a positive integer N from the input and returns the sum of the numbers from 1 to N, N included.

```
N = int(input('N: '))
sum = 0
while N>0:
    sum = sum + N
    N = N - 1
print(sum)
```

Write a code that takes a positive integer N from the input and returns the sum of the numbers from 1 to N, N included.

```
N = int(input('N: '))
sum = 0
while N>0:
    print('N: ' + str(N) + ' sum: ' + str(sum))
    sum = sum + N
    N = N - 1
print(sum)
```

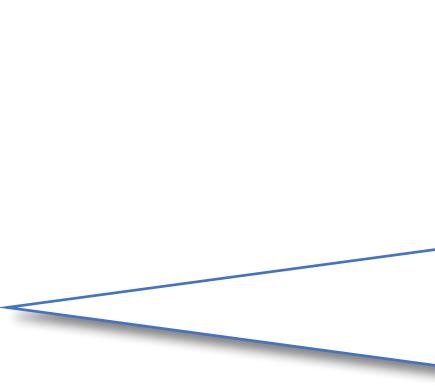
What's the output
for N is 5?

Write a code that takes a positive integer N from the input and returns the sum of the numbers from 1 to N, N included.

```
N = int(input('N: '))
sum = 0
while N>0:
    print('N: ' + str(N) + ' sum: ' + str(sum))
    sum = sum + N
    N = N - 1
print(sum)
```

```
N: 5
N: 5 sum: 0
N: 4 sum: 5
N: 3 sum: 9
N: 2 sum: 12
N: 1 sum: 14
15
```

Try this:



Write a code that takes a positive integer N from the input and returns the sum of the even numbers from 0 to N , N included.

Try this:

Write a code that takes a positive integer N from the input and returns the sum of the even numbers from 0 to N, N included.

```
N = int(input('N: '))
sum = 0
while N>0:
    if N%2==0:
        sum = sum + N
    N = N - 1
print(sum)
```

N: 5
6

N: 11
30

Alternatively...

```
N = int(input('N: '))
i, sum = 0, 0
while i<=N:
    sum = sum + i
    i = i + 2
print(sum)
```

Write a code that takes a positive integer N from the input and returns the sum of the even numbers from 0 to N, N included.

N: 5
6

N: 11
30

Be careful, small changes may have big effects...

```
N = int(input('N: '))
i, sum = 0, 0
while i<=N:
    sum = sum + i
    i = i + 2
print(sum)
```

N: 5
6

N: 11
30

```
N = int(input('N: '))
i, sum = 0, 0
while i<=N:
    i = i + 2
    sum = sum + i
print(sum)
```

what if I write it like this?

Be careful, small changes may have big effects...

```
N = int(input('N: '))
i, sum = 0, 0
while i<=N:
    sum = sum + i
    i = i + 2
print(sum)
```

N: 5
6

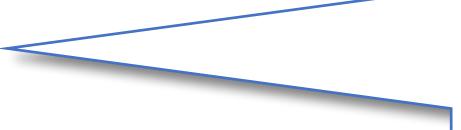
N: 11
30

```
N = int(input('N: '))
i, sum = 0, 0
while i<=N:
    i = i + 2
    sum = sum + i
print(sum)
```

what if I write it like this?

TODO figure out the problem
at home..

Try this:



Write a program that asks the user to input 5 integers, and then prints the largest odd number that was entered. If no odd number was entered, it should print 'None'.

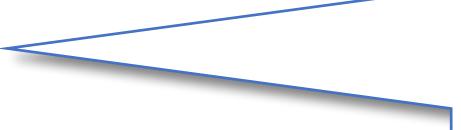
Input 1: 1 2 3 4 5

Input 2: -1 -2 -3 -4 -5

Output 1: 5

Output 2: -1

Try this:



Write a program that asks the user to input 5 integers, and then prints the largest odd number that was entered. If no odd number was entered, it should print 'None'.

TODO write on your own at home..

Lecture 6

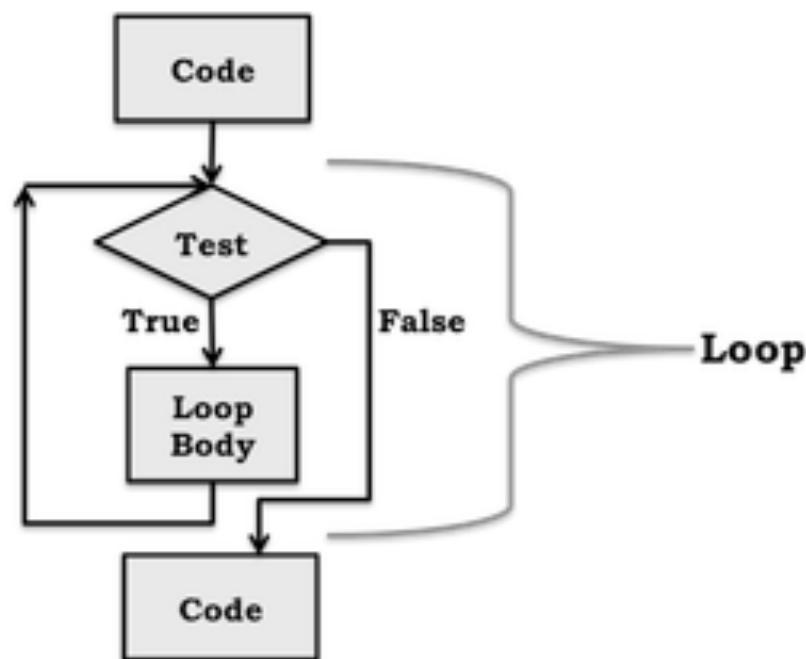
Iteration - for loops

Dr. Hacer Yalım Keleş

hkeles@ankara.edu.tr

Disclaimer: Content of this lecture slides are mostly from **Chapter 3** of Guttag's Book: "*Introduction to Computation and Programming Using Python*".

Flow chart for iteration



For loops

```
for <var> in <sequence>:  
    <body>
```

For loops

```
for <var> in <sequence>:  
    <body>
```

[<val1>, <val2>, ..., <valN>]

The evaluation continues until the sequence is consumed
Or a break statement is executed.

For loops

The evaluation continues until the sequence is consumed

Or a break statement is executed.

```
for <var> in <sequence>:  
    <body>
```

[<val1>, <val2>, ..., <valN>]

```
for i in [1,2,3,4,5]:  
    print(i)
```

For loops

The evaluation continues until the sequence is consumed

Or a break statement is executed.

```
for <var> in <sequence>:  
    <body>
```

[<val1>, <val2>, ..., <valN>]

```
for i in [1,2,3,4,5]:  
    print(i)
```

Output:

```
1  
2  
3  
4  
5
```

For loops break statement

The evaluation continues until the sequence is consumed

Or a break statement is executed.

```
for <var> in <sequence>:  
    <body>
```

→ [<val1>, <val2>, ..., <valN>]

```
for i in [1,2,3,4,5]:  
    if i%3==0:  
        break  
    print(i)
```

For loops

The evaluation continues until the sequence is consumed

Or a break statement is executed.

```
for <var> in <sequence>:  
    <body>
```

→ [<val1>, <val2>, ..., <valN>]

```
for i in [1,2,3,4,5]:  
    if i%3==0:  
        break  
    print(i)
```

Output:

```
1  
2
```

For loops

continue statement

Continue statement breaks a particular iteration,
loop goes on with the next iteration

```
for i in [1,2,3,4,5]:  
    if i%3==0:  
        continue  
    print(i)
```

Output:

```
1  
2  
4  
5
```

Generation of sequences:

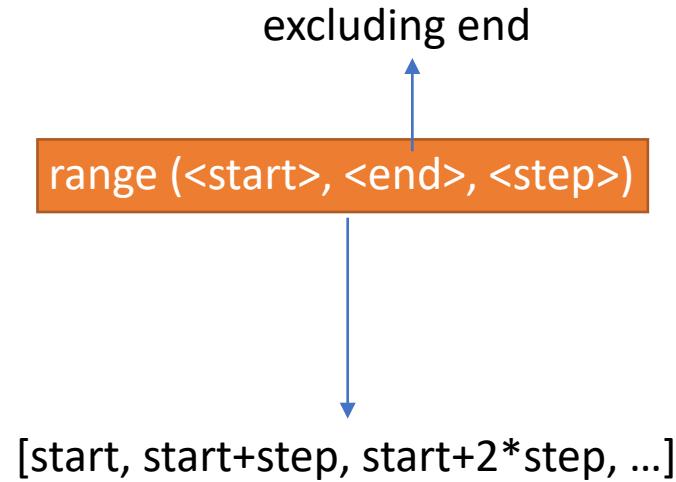
range function

range (<start>, <stop>, <step>)



[start, start+step, start+2*step, ...]

Generation of sequences: range function



start is 0 by default
step is 1 by default

range(3) → [0,1,2] → range(0,3)
range(1,5) → [1,2,3,4]
range(1,5,1) → [1,2,3,4]
range(3,10,2) → [3,5,7,9]
range(40,5,-10) → [40,30,20,10]

For loops

```
x = 4  
for i in range(0, x):  
    print i
```

For loops

```
x = 4  
for i in range(0, x):  
    print i
```

[0,1,2,3]

Output:

```
0  
1  
2  
3
```

Try this:

recap



Write a code that takes a positive integer N from the input and returns the sum of the even numbers from 0 to N, N included.

Try this:

```
N = int(input("N: "))
sum = 0
for i in range(N):
    if (i%2==0): sum = sum + i

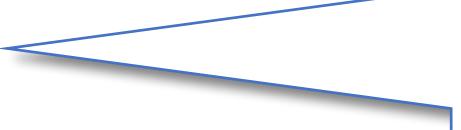
print(sum)
```

Write a code that takes a positive integer N from the input and returns the sum of the even numbers from 0 to N, N excluded.

N: 5
6

N: 11
30

Try this:



Write a program that asks the user to input 5 integers, and then prints the largest odd number that was entered. If no odd number was entered, it should print 'None'.

Input 1: 1 2 3 4 5

Input 2: -1 -2 -3 -4 -5

Output 1: 5

Output 2: -1

Try this:

```
print("Please input 5 integers:")
largest_odd_number = None
for i in range(5):
    number = int(input("integer "+str(i+1)+" : "))
    if number%2==1:
        if largest_odd_number == None:
            largest_odd_number = number
        elif number>largest_odd_number:
            largest_odd_number = number

print(largest_odd_number)
```

Write a program that asks the user to input 5 integers, and then prints the largest odd number that was entered. If no odd number was entered, it should print 'None'.

Try this:

```
print("Please input 5 integers:")
largest_odd_number = None
for i in range(5):
    number = int(input("integer "+str(i+1)+" : "))
    if number%2==1:
        if largest_odd_number == None:
            largest_odd_number = number
        elif number>largest_odd_number:
            largest_odd_number = number

print(largest_odd_number)
```

Write a program that asks the user to input 5 integers, and then prints the largest odd number that was entered. If no odd number was entered, it should print 'None'.

```
Please input 5 integers:
integer 1 : 1
integer 2 : 2
integer 3 : 4
integer 4 : 6
integer 5 : 8
1
```

Try this:

```
print("Please input 5 integers:")
largest_odd_number = None
for i in range(5):
    number = int(input("integer "+str(i+1)+" : "))
    if number%2==1:
        if largest_odd_number == None:
            largest_odd_number = number
        elif number>largest_odd_number:
            largest_odd_number = number

print(largest_odd_number)
```

```
Please input 5 integers:
integer 1 : -1
integer 2 : -2
integer 3 : -3
integer 4 : -4
integer 5 : -5
-1
```

Write a program that asks the user to input 5 integers, and then prints the largest odd number that was entered. If no odd number was entered, it should print 'None'.

Try this:

```
print("Please input 5 integers:")
largest_odd_number = None
for i in range(5):
    number = int(input("integer "+str(i+1)+" : "))
    if number%2==1:
        if largest_odd_number == None:
            largest_odd_number = number
        elif number>largest_odd_number:
            largest_odd_number = number

print(largest_odd_number)
```

```
Please input 5 integers:
integer 1 : -2
integer 2 : -4
integer 3 : 6
integer 4 : 8
integer 5 : 10
None
```

Write a program that asks the user to input 5 integers, and then prints the largest odd number that was entered. If no odd number was entered, it should print 'None'.

Try this:

```
n = 5
for i in range(n):
    print(i)
n = 3
```

Does changing *x* in the loop body effects the range of the generated sequence ?

Try this:

```
n = 5  
for i in range(n):  
    print(i)  
    n = 3
```

Does changing *x* in the loop body effects the range of the generated sequence ?

No! range is evaluated just before the first iteration and not reevaluated

Output:

```
0  
1  
2  
3  
4
```

Note that at the end of the for loop, *n* is 3!

Try this:

```
x = 2
for i in range(x):
    for j in range(x):
        print(i,j)
x = 4
```

What is the output of this sample then?

Exhaustive Enumeration:

Let's write a program that finds the proper divisors of a given positive integer

The **proper divisors** of the integer n are the positive **divisors** of n other than n itself.

Some Examples:

6 → 1, 2, 3

12 → 1, 2, 3, 4, 6

15 → 1, 3, 5

13 → 1

19 → 1

7 → 1

Any suggestions?

Exhaustive Enumeration:

Let's write a program that finds the proper divisors of a given positive integer

The **proper divisors** of the integer n are the positive **divisors** of n other than n itself.

Some Examples:

6 → 1, 2, 3

12 → 1, 2, 3, 4, 6

15 → 1, 3, 5

13 → 1

19 → 1

7 → 1

```
n = int(input())
for i in range(1,n):
    if n%i==0:
        print(i)
```

Exhaustive Enumeration:

Let's write a program that finds if a given number is a PRIME NUMBER

The **proper divisors** of a prime number is 1.

Some Examples:

6 → 1, 2, 3

12 → 1, 2, 3, 4, 6

15 → 1, 3, 5

13 → 1

19 → 1

7 → 1

Any suggestions?

Exhaustive Enumeration:

Let's write a program that finds the first even number that is divisible by 5 and 6, following a given number

Some Examples:

6 → 30

12 → 30

15 → 30

32 → 60

Any suggestions?

Exhaustive Enumeration:

Let's write a program that finds the square root of a given number using exhaustive enumeration, if it's a perfect square. Otherwise, report not found.

Some Examples:

25 —> 5

36 —> 6

Enumeration using strings:

Perfectly fine!

```
for c in "Hello":  
    print(c)
```

Output:

H
e
l
l
o

Enumeration using strings:

Compute the sum of the digits in a given string if there is any..

Some Examples:

“123” —> 6

“COM1001” —> 2

“COM10012345B”—>16

Enumeration using strings:

Compute the sum of the digits in a given string if there is any..

Some Examples:

"123" —> 6

"COM1001" —> 2

"COM10012345B"—>16

Let me first explain representation of characters
and ASCII Table...

We'll come back to this, remind me if I forget.

Chaotic Enumeration:

```
def chaotic(x, lc):
    print("----- chaotic -----")
    for i in range(lc):
        x = 3.9*x*(1-x)
        print(x)
```

Chaotic Enumeration-

A story about little precision issues and serious consequences..

```
def chaotic(x, lc):
    print("----- chaotic -----")
    for i in range(lc):
        x = 3.9*x*(1-x)
        print(x)
```

x=0.25

```
----- chaotic -----
0.73125
0.76644140625
0.6981350104385375
0.8218958187902304
0.5708940191969317
0.9553987483642099
0.166186721954413
0.5404179120617926
0.9686289302998042
0.11850901017563877
```

0.255

```
----- chaotic -----
0.7409024999999999
0.7486673434256251
0.7338417540232046
0.7617403328938691
0.7078187927327657
0.8065642626096735
0.6084715762732898
0.9291122768490727
0.2568643500438786
0.7444517173096161
```

x=0.26

```
----- chaotic -----
0.75036
0.73054749456
0.7677066257332165
0.6954993339002887
0.8259420407337192
0.5606709657211202
0.9606442322820199
0.14744687593470315
0.49025454937601765
0.9746296021493285
```

Precision issues with floats Cntd.

Another story about little precision issues and serious consequences..

```
def chaoticAll(x, lc):
    x1 = x
    x2 = x
    x3 = x
    for i in range(lc):
        x1 = 3.9 * x1 * (1 - x1)
        x2 = 3.9*x2-3.9*x2*x2
        x3 = 3.9 * (x3 - x3 * x3)
    print(x1, x2, x3, sep='\t')
```

Same mathematical eqn.
Written using 3 different ways

Run this code using this and check the results:

```
x = float(input())
chaoticAll(x, 100)
```

Let me also explain representations of floats...

COM1001

Data Representation Details

Dr. Hacer Yalım Keleş

Credit: Most of the slides from this lecture note is borrowed and adapted from ceng111 Lecture Notes from METU.

Basic Data Types

Integers:

- Full support from the CPU
- Fast processing

Floating Points:

- Support from the CPU with some precision loss
- Slower compared to integer processing due to “interpretation”

Recap and go into details...

BINARY REPRESENTATIONS

Binary Representation of Numeric Information

- Decimal numbering system

- Base-10
 - Each position is a power of 10

$$3052 = 3 \times 10^3 + 0 \times 10^2 + 5 \times 10^1 + 2 \times 10^0$$

- Binary numbering system

- Base-2
 - Uses ones and zeros
 - Each position is a power of 2

$$1101 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

Decimal-to-binary Conversion

Divide the number until zero:

$$35 / 2 = 17 \times 2 + 1$$

$$17 / 2 = 8 \times 2 + 1$$

$$8 / 2 = 4 \times 2 + 0$$

$$4 / 2 = 2 \times 2 + 0$$

$$2 / 2 = 1 \times 2 + 0$$

Therefore, 35 has the binary representation: 100011

Binary Representation of Numeric Information (continued)

- Representing integers

- Decimal integers are converted to binary integers
- Question: given k bits, what is the value of the largest integer that can be represented?

$$2^k - 1$$

Ex: given 4 bits, the largest is $2^4 - 1 = 15$

- Signed integers must also represent the sign (positive or negative) - ***Sign/Magnitude notation***

Binary Representation of Numeric Information (continued)

Sign/magnitude notation

$$1 \ 101 = -5$$

$$0 \ 101 = +5$$

Problems:

- Two different representations for 0:

$$1 \ 000 = -0$$

$$0 \ 000 = +0$$

- Addition & subtraction require a watch for the sign! Otherwise, you get wrong results:

$$0 \ 010 (+2) + 1 \ 010 (-2) = 1 \ 100 (-4)$$

Arithmetic in Computers is Modular

Let's add two numbers in binary
(Assume that there is no sign bit)

$$\begin{array}{r} \begin{array}{|c|c|c|c|} \hline 1 & 0 & 1 & 1 \\ \hline 1 & 1 & 1 & 0 \\ \hline \end{array} \\ + \\ \hline \end{array} \quad \begin{array}{r} \xrightarrow{\hspace{2cm}} (11)_{10} \\ \xrightarrow{\hspace{2cm} + } (14)_{10} \\ \hline \xrightarrow{\hspace{2cm}} (9)_{10} \end{array}$$

■ In other words:

Numbers larger than or equal to 16 (2^4) are discarded in a 4-bit representation.

Therefore, $11 + 14$ yields 9 in this 4-bit representation.

This is actually modular arithmetic:

$$11 + 14 \bmod 16 = 9 \bmod 16$$

Binary Representation of Numeric Information (continued)

- ➊ Two's complement instead of sign-magnitude representation

- Positive numbers have a leading 0.
 - 5 => 0101
 - The representation for negative numbers is found by subtracting the absolute value from 2^N for an N-bit system:
 - $-5 \Rightarrow 2^4 - 5 = 16 - 5 = (11)_{10} \Rightarrow (1011)_2$

- ➋ Advantages:

- 0 has a single representation: +0 = 0000, -0 = 0000
 - Arithmetic works fine without checking the sign bit:
 - $1011 (-5) + 0110 (6) = 0001 (1)$
 - $1011 (-5) + 0011 (3) = 1110 (-2)$

Binary Representation of Numeric Information (continued)

- Shortcut to convert from “two’s complement” :
 - If the leading bit is zero, no need to convert.
 - If the leading bit is one, invert the number and add 1.
- What is our range?
 - With 2’s complement we can represent numbers from -2^{N-1} to $2^{N-1} - 1$ using N bits.
 - 8 bits: -128 to +127.
 - 16 bits: -32,768 to +32,767.
 - 32 bits: -2,147,483,648 to +2,147,483,647.

Binary Number	Decimal Value	Value in Two's Complement
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

Binary Representation of Numeric Information (continued)

Example:

- We want to compute: $12 - 6$
 - $12 \Rightarrow 01100$ invert add one
 - $-6 \Rightarrow -(00110) \Rightarrow (11001)+1 \Rightarrow (11010)$
- Two's complement notation for -6

$$12 - 6 =$$

$$\begin{array}{r} 01100 \rightarrow (12)_{10} \\ + 11010 \rightarrow (-6)_{10} \\ \hline \end{array}$$

So, addition and subtraction operations are simpler in the Two's Complement representation

$$00110 \rightarrow (6)_{10}$$

Binary Representation of Numeric Information (continued)

Due to its advantages, two's complement is the most common way to represent integers on computers.

Binary Representation of Real Numbers

Conversion of the digits after the dot into binary:

- 1st Way:

- $0.375 \rightarrow 0 \times 1/2 + 1 \times 1/4 + 1 \times 1/8 \rightarrow 011$

- 2nd Way:

- Multiply by 2 and get the integer part until we get '0' after the dot:

- $0.375 \times 2 = 0.750 = 0 + 0.750$
 - $0.750 \times 2 = 1.500 = 1 + 0.500$
 - $0.500 \times 2 = 1.000 = 1 + 0.000$

Binary Representation of Real Numbers

Approach 1: Use fixed-point

- Similar to integers, except that there is a decimal point.

e.g: using 8 bits:

1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---

•
↑

Assumed decimal point

$$\begin{aligned} &= 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} + 1 \times 2^{-4} \\ &= 15.9375 \end{aligned}$$

Binary Representation of Real Numbers

Location of the decimal point changes the value of the number.

E.g.: using 8 bits:

1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---

•
↑

$$= 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\ 1 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} = 31.875$$

Assumed decimal point

Binary Representation of Real Numbers

Problems with fixed-point:

- Limited in the maximum and minimum values that can be represented.
- For instance, using 32-bits, reserving 1-bit for the sign and putting the decimal point after 16 bits from the right, the maximum positive value that can be stored is slightly less than 2^{15} .
- Allowing larger values gives away from the precision (the decimal part).

Binary Representation of Real Numbers

Solution: Use scientific notation: $a \times 2^b$ (or $\pm M \times B^{\pm E}$)

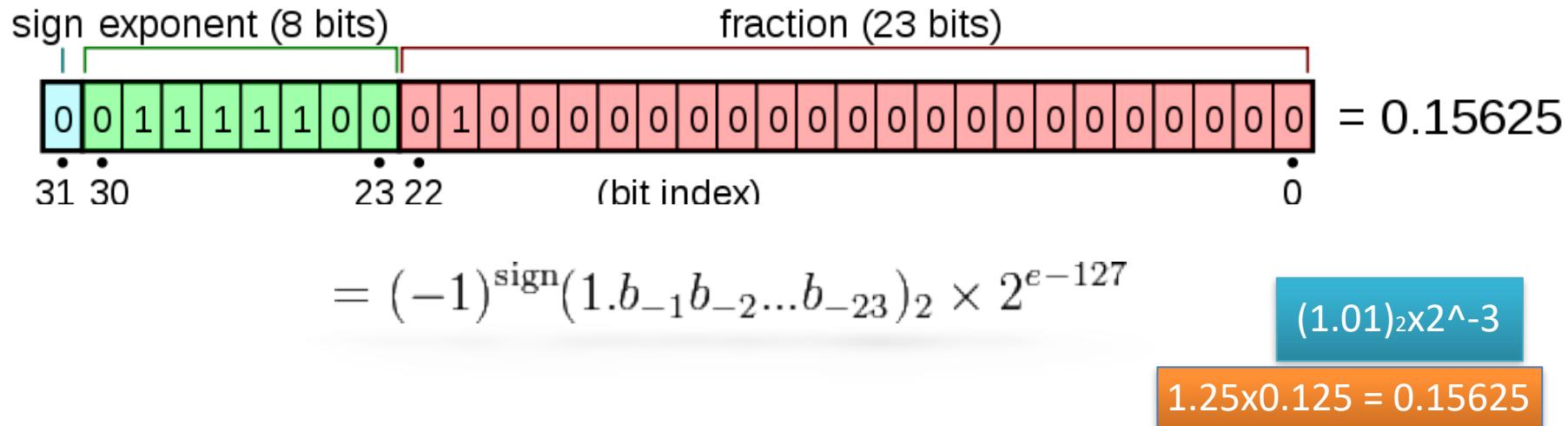
- Example: 5.75
 - $5 \rightarrow 101$
 - $0.75 \rightarrow \frac{1}{2} + \frac{1}{4} \rightarrow 2^{-1} + 2^{-2} \rightarrow (0.11)_2$
 - $5.75 \rightarrow (101.11)_2 \times 2^0$
- Number is then normalized so that the first significant digit is immediately to the left of the binary point
 - Example: 1.0111×2^2
- We take and store the **mantissa** and the **exponent**.

Binary Representation of Real Numbers

This needs some standardization for:

- where to put the decimal point
- how to represent negative numbers
- how to represent numbers less than 1

IEEE 32bit Floating-Point Number Representation



- $M \times 2^E$
 - Exponent (E): 8 bits
 - Add 127 to the exponent value before storing it
 - **E can be 0 to 255 with 127 representing the real zero.**
 - Fraction (M - Mantissa): 23 bits

IEEE 32bit Floating-Point Number Representation

Example: 12.375

- The digits before the dot:
 - $(12)_{10} \rightarrow (1100)_2$
- The digits after the dot:
 - 1st Way: $0.375 \rightarrow 0 \times \frac{1}{2} + 1 \times \frac{1}{4} + 1 \times \frac{1}{8} \rightarrow 011$
 - 2nd Way: Multiply by 2 and get the integer part until 0:
 - $0.375 \times 2 = 0.750 = 0 + 0.750$
 - $0.750 \times 2 = 1.50 = 1 + 0.50$
 - $0.50 \times 2 = 1.0 = 1 + 0.0$
- $(12.375)_{10} = (1100.011)_2$
- NORMALIZE (move the point): $(1100.011)_2 = (1.100011)_2 \times 2^3$
- Exponent: 3, adding 127 to it, we get 130= $(1000\ 0010)_2$
- Fraction: 100011
- Then our number is: 0 $10000010\ 100011000000000000000000$

Problems due to precision loss

- ➊ $1.0023 - 1.0567$
 - Result: -0.05440000000000004
- ➋ $1000.0023 - 1000.0567$
 - Result: -0.05439999999986903
- ➌ Why?
 - Since the floating point representation is based on shifting the bits in the mantissa, they are not equivalent in a PC
- ➍ $\pi = 3.1415926535897931\ldots$
 - $\sin(\pi)$ should be zero
 - But it is not: $1.2246467991473532 \times 10^{-16}$

Problems due to precision loss

Associativity in mathematics:

$$(a+b) + c = a + (b + c)$$

This does not hold in floating-point arithmetic:

set $a = 1234.567$, $b = 45.67834$ and $c = 0.0004$:
 $(a + b) + c$ will produce 1280.2457399999998,
 $a + (b + c)$ will produce 1280.2457400000001.

Binary Representation of Textual Information

- Characters are mapped onto binary numbers
 - ASCII (**American Standard Code for Information Interchange**) code set
 - Originally: 7 bits per character; 128 character codes
 - Unicode code set
 - 16 bits per character
 - UTF-8 (**Universal Character Set Transformation Format**) code set.
 - Variable number of 8-bits.

Binary Representation of Textual Information (cont'd)

ASCII
7 bits long

Decimal	Binary	Val.
48	00110000	0
49	00110001	1
50	00110010	2
51	00110011	3
52	00110100	4
53	00110101	5
54	00110110	6
55	00110111	7
56	00111000	8
57	00111001	9
58	00111010	:
59	00111011	;
60	00111100	<
61	00111101	=
62	00111110	>
63	00111111	?
64	01000000	@
65	01000001	A
66	01000010	B

Hex.	Unicode	Charac.
0x30	0x0030	0
0x31	0x0031	1
0x32	0x0032	2
0x33	0x0033	3
0x34	0x0034	4
0x35	0x0035	5
0x36	0x0036	6
0x37	0x0037	7
0x38	0x0038	8
0x39	0x0039	9
0x3A	0x003A	:
0x3B	0x003B	;
0x3C	0x003C	<
0x3D	0x003D	=
0x3E	0x003E	>
0x3F	0x003F	?
0x40	0x0040	@
0x41	0x0041	A
0x42	0x0042	B

Unicode
16 bits long

Partial
listings
only!

How about a text?

- ➊ Text in a computer has two alternative representations:
 - ➌ A fixed-length number representing the length of the text followed by the binary values of the characters in the text.
 - Ex: “ABC” =>
00000011 **01000001** 01000010 **01000011** (3 ‘A’ ‘B’ ‘C’)
 - ➍ Binary values of the characters in the text ended with a unique marker, like “00000000” which has no value in the ASCII table.
 - Ex: “ABC” =>
01000001 01000010 **01000011** 00000000 (‘A’ ‘B’ ‘C’ END)

Recap...

PYTHON SIDE

Integers in Python

Python provides **int** type.

```
>>> type(3)
<type 'int'>
>>> type(3+4)
<type 'int'>
>>>
```

As opposed to other
Programming languages,
int type is unlimited
in Python.

Floating Points in Python

Python provides **float** type.

```
>>> type(3.4)
<type 'float'>
>>>
```

```
>>> 3.4+4.3
7.7
>>> 3.4 / 4.3
0.79069767441860461
```

Simple Operations with Numerical Values in Python

Operator	Operator Type	Description
+	Binary	Addition of two operands
-	Binary	Subtraction of two operands
-	Unary	Negated value of the operand
*	Binary	Multiplication of two operands
/	Binary	Division of two operands
**	Binary	Exponentiation of two operands (Ex: $x^{**}y = x^y$)

abs(x): Absolute value of x.

round(Float): Rounded value of float.

int(Number), float(Number): Conversion between numerical values of different types.

Characters

We need characters to represent textual data:

Characters: 'A', ..., 'Z', '0', ..., '9', 'a', ..., 'z' ... etc.

Computer uses the ASCII table for converting characters to binary numbers:

Characters in Python

- Python does not have a separate data type for characters!
- However, one character strings can be treated like characters:
`ord(<One_Char_String>)` : returns the ASCII value of the character in One_Char_String.
Ex: `ord("A") —> 65.`
`chr(<ASCII_value>)` : returns the character in a string that has the ASCII value ASCII_value:
Ex: `chr(66) —> "B"`

Boolean Values

- ➊ The CPU often needs to compare numbers, or data:
 - $3 >? 4$
 - $125 =? 1000/8$
 - $3 \leq? 12345.34545/12324356.0$

- ➋ We have the truth values for representing the answers to such comparisons:
 - If correct: True, 1
 - If not correct: False, 0
 - Conversions from numeric values to Boolean
 - Any non-zero value is considered as True, e.g. `bool(4), bool(-300)`
 - Zero is interpreted as False, e.g. `bool(0), bool(0.0)`

Boolean Values in Python

Python provides the **bool** data type for boolean values.

bool data type can take True or False as values.

```
>>> 3 > 4  
False  
>>> type(3 > 4)  
<type 'bool'>
```

```
>>> True == 2  
False  
>>> True == 1  
True  
>>> False == 0  
True  
>>> False == 1  
False
```

not True

not operator: not 4 > 3

Try this:

```
>>> True == bool(2)
```

Lecture 8

Functions

Dr. Hacer Yalım Keleş

hkeles@ankara.edu.tr

Disclaimer: Content of this lecture slides are mostly from **Chapter 4** of Guttag's Book: "*Introduction to Computation and Programming Using Python*". Some slide contents are taken from https://python-textbook.readthedocs.io/en/1.0/Variables_and_Scope.html

Function definition

```
def <func_name>( <param_list> ):  
    <func_body>
```

Function definition

```
def <func_name>(<param_list>):  
    <func_body>
```

Function definition:

```
def max(x, y):  
    if x > y:  
        return x  
    else:  
        return y
```

Formal parameters

Function definition

```
def <func_name>(<param_list>):  
    <func_body>
```

Function definition:

```
def max(x, y):  
    if x > y:  
        return x  
    else:  
        return y
```

Formal parameters

Function call:

```
max(3, 4)
```

Binds x to the argument 3,
y to the argument 4.

Function definition

```
def <func_name>(<param_list>):  
    <func_body>
```

Function definition:

```
def max(x, y):  
    if x > y:  
        return x  
    else:  
        return y
```

Function call:

```
max(3, 4)
```

- Function call is an expression, i.e. it has a value.
- The value of a function call is the value returned by the invoked function
Example: max(3,5)*max(4,2) → 20
- Function call terminates after the **return** statement

When a function is called;

(1)

The expressions that make up the actual parameters are evaluated, and the formal parameters of the function are bound to the resulting values. For example, the invocation `max(3+4, z)` will bind the formal parameter `x` to 7 and the formal parameter `y` to whatever value the variable `z` has when the invocation is evaluated.

When a function is called;

(2)

The **point of execution** (the next instruction to be executed) moves from the point of invocation to the first statement in the body of the function.

When a function is called;

(3)

The code in the body of the function is executed until either a `return` statement is encountered, in which case the value of the expression following the `return` becomes the value of the function invocation, or there are no more statements to execute, in which case the function returns the value `None`. (If no expression follows the `return`, the value of the invocation is `None`.)

When a function is called;

(4)

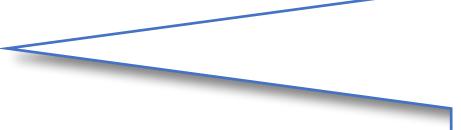
The value of the invocation is the returned value.

When a function is called;

(5)

The point of execution is transferred back to the code immediately following the invocation.

Try this:



Write a function `isIn` that accepts two strings as arguments and returns `True` if either string occurs anywhere in the other, and `False` otherwise.

Keyword arguments & default values

In addition to the positional parameter bounding,
formal parameters can also be bound to the actual arguments
using the name of the formal parameters during function call.

Keyword arguments & default values

In addition to the positional parameter binding, formal parameters can also be bound to the actual arguments using the name of the formal parameters during function call.

Example

```
def printName(firstName, lastName, reverse):
    if reverse:
        print(lastName + ', ' + firstName)
    else:
        print(firstName + ', ' + lastName)
```

Keyword arguments & default values

Example

```
def printName(firstName, lastName, reverse):
    if reverse:
        print(lastName + ', ' + firstName)
    else:
        print(firstName + ', ' + lastName)
```

Alternative function calls

```
printName('Olga', 'Puchmajerova', False)
printName('Olga', 'Puchmajerova', reverse = False)
printName('Olga', lastName = 'Puchmajerova', reverse = False)
printName(lastName='Puchmajerova', firstName='Olga', reverse=False)
```

Keyword arguments

Though the keyword arguments can appear in any order in the list of actual parameters, it is not legal to follow a keyword argument with a non-keyword argument. Therefore, an error message would be produced by

Keyword arguments

Though the keyword arguments can appear in any order in the list of actual parameters, it is not legal to follow a keyword argument with a non-keyword argument. Therefore, an error message would be produced by

```
printName('Olga', lastName = 'Puchmajerova', False)
```

Keyword arguments

Though the keyword arguments can appear in any order in the list of actual parameters, it is not legal to follow a keyword argument with a non-keyword argument. Therefore, an error message would be produced by

```
printName('Olga', lastName = 'Puchmajerova', False)
```



Non-keyword arg. after a keyword arg.

Default values

Default values allow programmers to call a function with fewer than the specified number of arguments.

```
def printName(firstName, lastName, reverse=False):
    if reverse:
        print(lastName + ', ' + firstName)
    else:
        print(firstName + ', ' + lastName)
```

Default values

Default values allow programmers to call a function with fewer than the specified number of arguments.

```
def printName(firstName, lastName, reverse=False):
    if reverse:
        print(lastName + ', ' + firstName)
    else:
        print(firstName + ', ' + lastName)
```

```
printName('Olga', 'Puchmajerova')
printName('Olga', 'Puchmajerova', True)
printName('Olga', 'Puchmajerova', reverse = True)
```

Default values

Default values allow programmers to call a function with fewer than the specified number of arguments.

```
def printName(firstName, lastName, reverse=False):
    if reverse:
        print(lastName + ', ' + firstName)
    else:
        print(firstName + ', ' + lastName)
```

```
printName('Olga', 'Puchmajerova')
printName('Olga', 'Puchmajerova', True)
printName('Olga', 'Puchmajerova', reverse = True)
```

Output:

```
Olga Puchmajerova
Puchmajerova, Olga
Puchmajerova, Olga
```

Scope

Variables

Recap

- A variable is a label for a location in memory. Storing a new value erases the old.
- Each variable is referred to by name
- In statically typed languages, variables have predetermined types, and a variable can only be used to hold values of that type.
- In Python, we may reuse the same variable to store values of any type.

Variables & Scope

- Not all variables are accessible from all parts of our program,
- Not all variables exist for the same amount of time.
- Where a variable is accessible and how long it exists depend on how it is defined.
- We call the part of a program where a variable is accessible its **scope**,
- We call the duration for which the variable exists its ***lifetime***.

Variables & Scope

- A variable which is defined in the main body of a file is called a *global* variable.
 - It will be visible throughout the file, and also inside any file which imports that file.
- Global variables can have unintended consequences because of their wide-ranging effects – that is why we should almost never use them.
- Only objects which are intended to be used globally, like **functions and classes**, should be put in the global namespace.

Variables & Scope

- A variable which is defined in the main body of a file is called a *global* variable.
 - It will be visible throughout the file, and also inside any file which imports that file.
- Global variables can have unintended consequences because of their wide-ranging effects – that is why we should almost never use them.
- Only objects which are intended to be used globally, like **functions and classes**, should be put in the global namespace.

Variables & Scope

- A variable which is defined inside a function is *local* to that function.
 - It is accessible from the point at which it is defined until the end of the function, and exists for as long as the function is executing.
- The parameter names in the function definition behave like local variables, but they contain the values that we pass into the function when we call it.
- When we use the assignment operator (=) inside a function, its default behaviour is to create a new local variable – unless a variable with the same name is already defined in the local scope.

```
# This is a global variable
a = 0

if a == 0:
    # This is still a global variable
    b = 1

def my_function(c):
    # this is a local variable
    d = 3
    print(c)
    print(d)

# Now we call the function, passing the value 7 as the first and only parameter
my_function(7)

# a and b still exist
print(a)
print(b)

# c and d don't exist anymore -- these statements will give us name errors!
print(c)
print(d)
```

Scope

```
a = 0

def my_function():
    print(a)

my_function()
```

Scope

```
a = 0

def my_function():
    a = 3
    print(a)

my_function()

print(a)
```

Scope

```
a = 0

def my_function():
    a = 3
    print(a)

my_function()

print(a)
```

By default, the assignment statement creates variables in the local scope.

Scope

What if we really want to modify a global variable from inside a function?
We can use the `global` keyword

```
a = 0

def my_function():
    a = 3 ←
    print(a)

my_function()

print(a)
```

local var a

Output:

```
3
0
```

```
a = 0

def my_function():
    global a
    a = 3
    print(a)

my_function()

print(a)
```

global var a

Output:

```
3
3
```

We may not refer to both a global variable and a local variable by the same name inside the same function.

```
a = 0

def my_function():
    print(a)
    a = 3
    print(a)

my_function()
```

We may not refer to both a global variable and a local variable by the same name inside the same function.

```
a = 0

def my_function():
    print(a)
    a = 3
    print(a)

my_function()
```

```
-----  
UnboundLocalError                                Traceback (most recent call last)
<ipython-input-4-1db3abec19a4> in <module>
      6     print(a)
      7
----> 8 my_function()

<ipython-input-4-1db3abec19a4> in my_function()
      2
      3 def my_function():
----> 4     print(a)
      5     a = 3
      6     print(a)

UnboundLocalError: local variable 'a' referenced before assignment
```

Note that it is usually very **bad practice** to access global variables from inside functions, and even worse practice to modify them. This makes it difficult to arrange our program into logically encapsulated parts which do not affect each other in unexpected ways.

If a function needs to access some external value, we should pass the value into the function as a parameter.

```
def my_function(a):
    b = a - 2
    return b

c = 3

if c > 2:
    d = my_function(5)
    print("d:",d)

    for i in range(3):
        d += i
    print("i:",i)

print("i:",i)
print("d:",d)
```

What are the scopes of variables in this example?

What is the output?

Mutable and Immutable Types

Some *values* in python can be modified, and some cannot.

This does not ever mean that we can't change the value of a variable – but if a variable contains a value of an *immutable type*, we can only assign it a *new value*.

—> We cannot *alter the existing value* in any way.

Mutable and Immutable Types

Integers, floating-point numbers and strings are all immutable types

when we changed the values of existing variables we used the assignment operator to assign them new values:

```
a = 3  
a = 2
```

```
b = "jane"  
b = "bob"
```

Even this operator does not change the value in place; it also assigns a new value:

```
total += 4
```

Scope

```
def f(x):  
    y = 1  
    x = x + y  
    print("x in f(x):", x)  
    return x
```

Function scope

x in f() is different from x outside the function scope

```
x = 2  
y = 5  
z = f(x)  
print("x :", x)  
print("y :", y)  
print("z :", z)
```

Scope

```
def f(x):
    y = 1
    x = x + y
    print("x in f(x):", x)
    return x
```

At the call of `f(x)`, the formal parameter is locally bound to the value of the actual parameter `x`

```
x = 2
y = 5
z = f(x)
print("x :", x)
print("y :", y)
print("z :", z)
```

function local scope

Global scope

```
x in f(x): 3
x : 2
y : 5
z : 3
```

Scope

```
def f(x):
    y = 1
    x = x + y
    print("x in f(x):", x)
    return x
```

At the call of `f(x)`, the formal parameter is locally bound to the value of the actual parameter `x`

function scope

```
x = 2
y = 5
z = f(x)
print("x :", x)
print("y :", y)
print("z :", z)
```

The formal parameter `x` and the local variable `y` exists only within the scope of `f`.

outer scope

```
x in f(x): 3
x : 2
y : 5
z : 3
```

Function (stack) frame

- At top level, i.e., the level of the shell, a **symbol table** keeps track of all names defined at that level and their current bindings.
- When a function is called, a new symbol table (sometimes called a **stack frame**) is created. This table keeps track of all names defined within the function (including the formal parameters) and their current bindings. If a function is called from within the function body, yet another stack frame is created.
- When the function completes, its stack frame goes away.

```
def f(x):
    def g():
        x = 'abc'
        print("g::x =", x)

    def h():
        z = x
        print("h::z =", z)

    x = x + 1
    print("f1::x =", x)
    h()
    g()
    print("f1::x =", x)
    return g

x = 3
z = f(x)
print("global::x =", x)
print("global::z =", z)
z()
```

func g is in the local scope of the func f

func h is in the local scope of the func

var x is in the local scope of the func f, i.e. formal parameter

function object g is returned

var x is in the global scope

var z is in the global scope

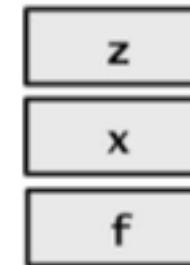
```
def f(x):
    def g():
        x = 'abc'
        print("g::x =", x)

    def h():
        z = x
        print("h::z =", z)

    x = x + 1
    print("f1::x =", x)
    h()
    g()
    print("f1::x =", x)
    return g

x = 3
z = f(x)
print("global::x =", x)
print("global::z =", z)
z()
```

STACK FRAME



1

```

def f(x):
    def g():
        x = 'abc'
        print("g::x =", x)

    def h():
        z = x
        print("h::z =", z)

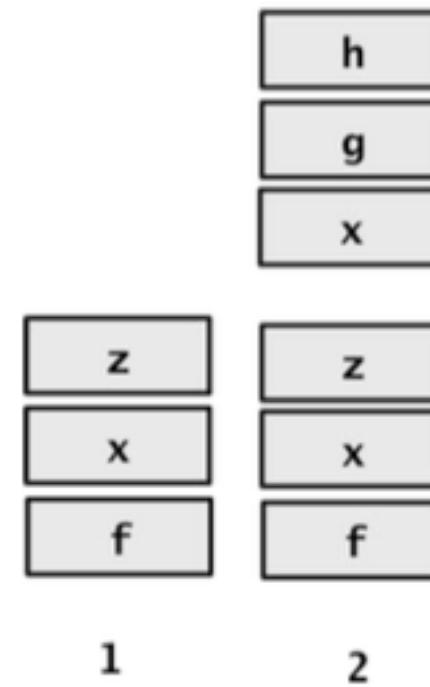
    x = x + 1
    print("f1::x =", x)
    h()
    g()
    print("f1::x =", x)
    return g

x = 3
z = f(x) ←
print("global::x =", x)
print("global::z =", z)
z()

```

(2)

STACK FRAME



```

def f(x):
    def g():
        x = 'abc'
        print("g::x =", x)

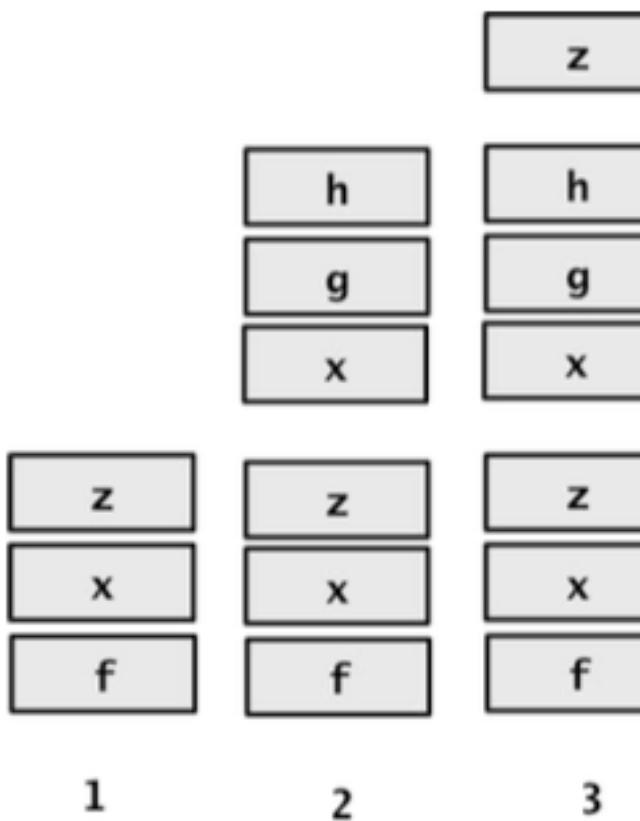
    def h():
        z = x
        print("h::z =", z)

        x = x + 1
        print("f1::x =", x)
        h()      ←
        g()
        print("f1::x =", x)
        return g

    x = 3
    z = f(x)
    print("global::x =", x)
    print("global::z =", z)
    z()

```

STACK FRAME



```

def f(x):
    def g():
        x = 'abc'
        print("g::x =", x)

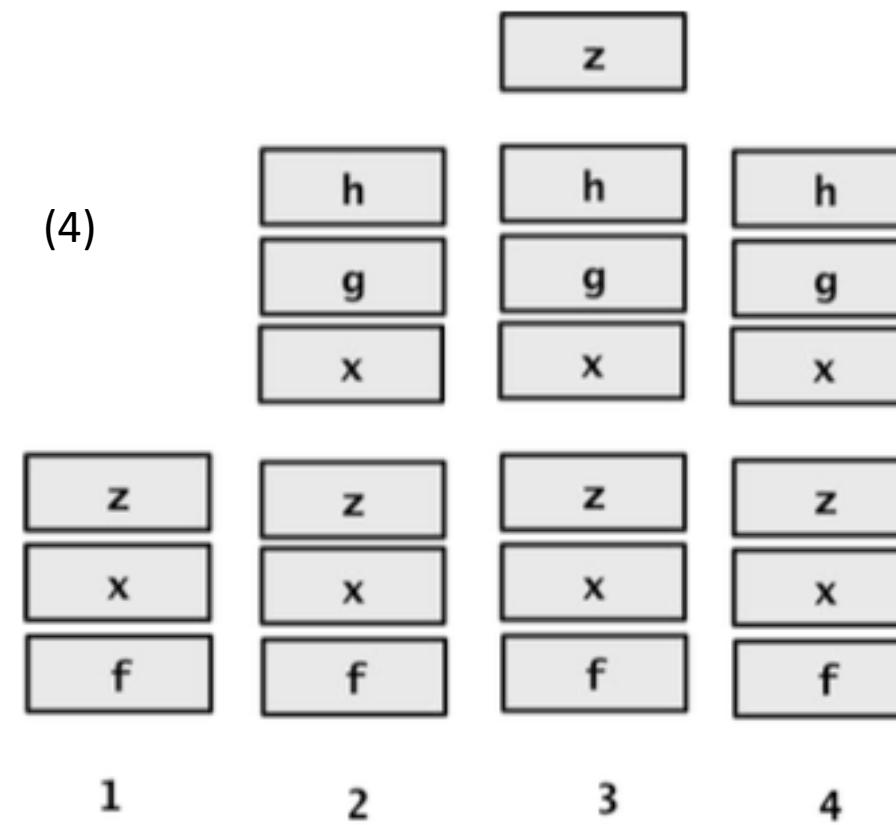
    def h():
        z = x
        print("h::z =", z)

    x = x + 1
    print("f1::x =", x)
    h()
    g()
    print("f1::x =", x)
    return g

x = 3
z = f(x)
print("global::x =", x)
print("global::z =", z)
z()

```

STACK FRAME



```

def f(x):
    def g():
        x = 'abc'
        print("g::x =", x)

    def h():
        z = x
        print("h::z =", z)

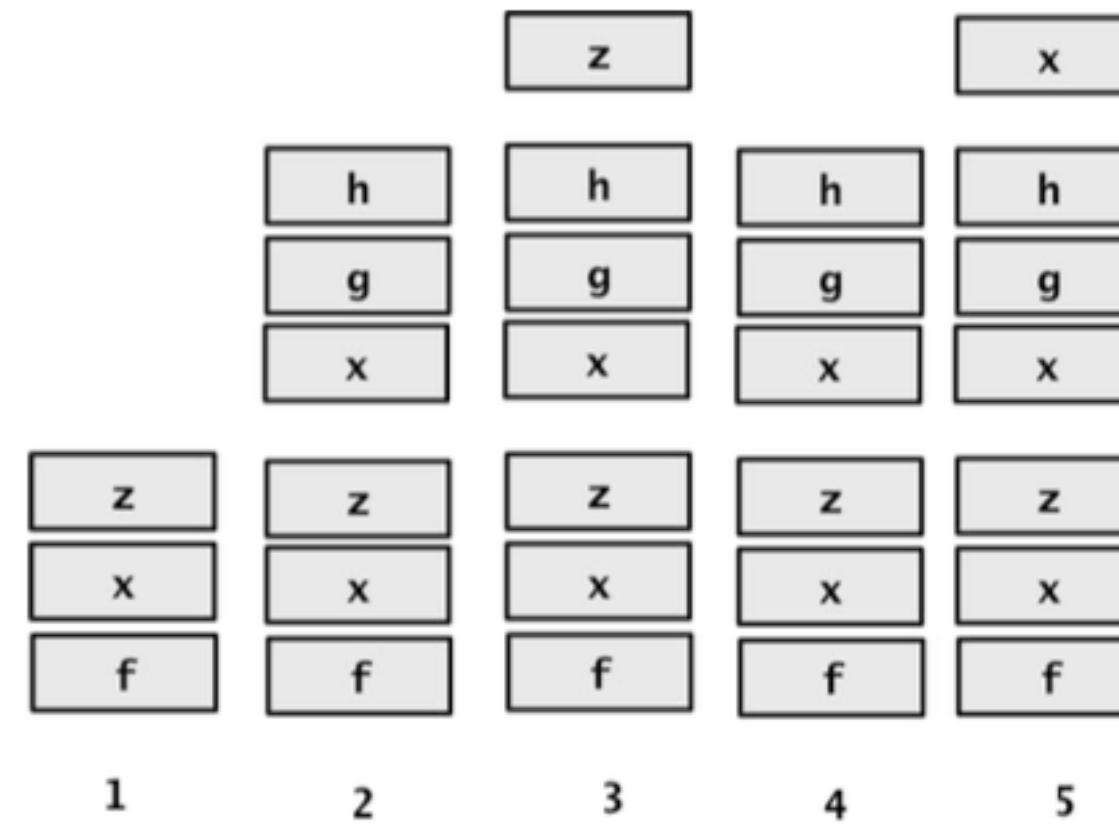
        x = x + 1
        print("f1::x =", x)
        h()
        g() ←
        print("f1::x =", x)
        return g

    x = 3
    z = f(x)
    print("global::x =", x)
    print("global::z =", z)
    z()

```

(5)

STACK FRAME



```

def f(x):
    def g():
        x = 'abc'
        print("g::x =", x)

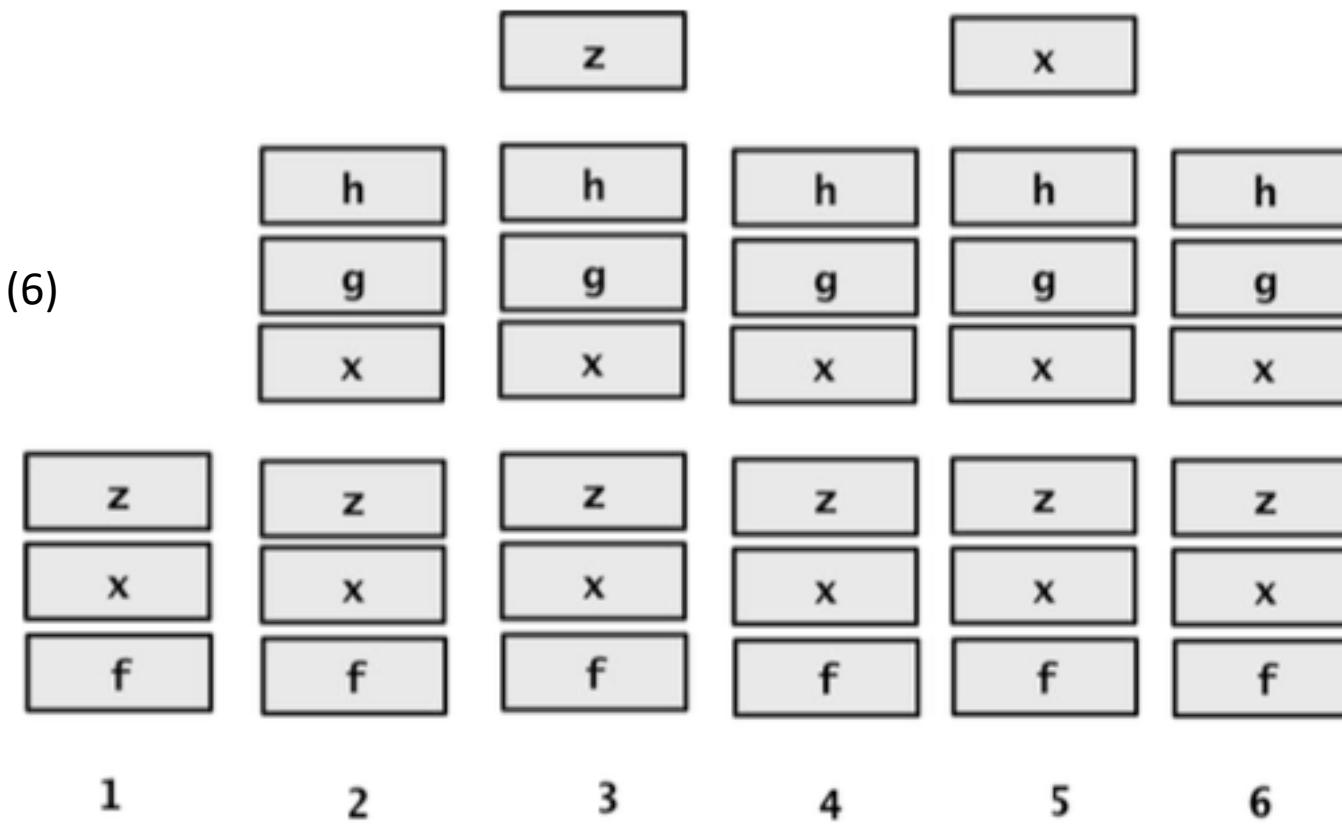
    def h():
        z = x
        print("h::z =", z)

        x = x + 1
        print("f1::x =", x)
        h()
        g()
        print("f1::x =", x)
        return g

    x = 3
    z = f(x)
    print("global::x =", x)
    print("global::z =", z)
    z()

```

STACK FRAME



```

def f(x):
    def g():
        x = 'abc'
        print("g::x =", x)

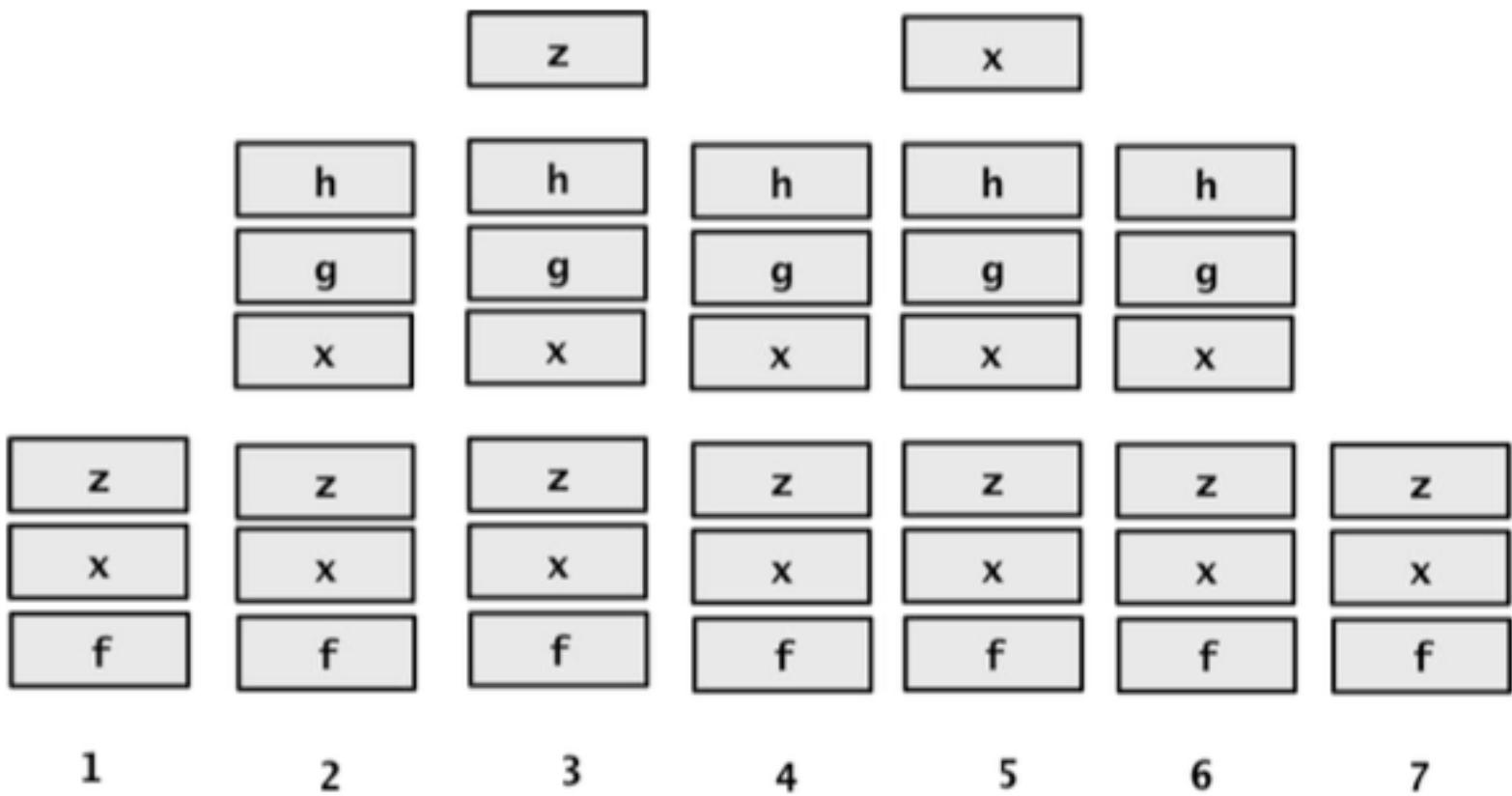
    def h():
        z = x
        print("h::z =", z)

        x = x + 1
        print("f1::x =", x)
        h()
        g()
        print("f1::x =", x)
        return g

    x = 3
    z = f(x) →(7)
    print("global::x =", x)
    print("global::z =", z)
    z()

```

STACK FRAME



```

def f(x):
    def g():
        x = 'abc'
        print("g::x =", x)

    def h():
        z = x
        print("h::z =", z)

        x = x + 1
        print("f1::x =", x)
        h()
        g()
        print("f1::x =", x)
        return g

    x = 3
    z = f(x) →(7)
    print("global::x =", x)
    print("global::z =", z)
    z()

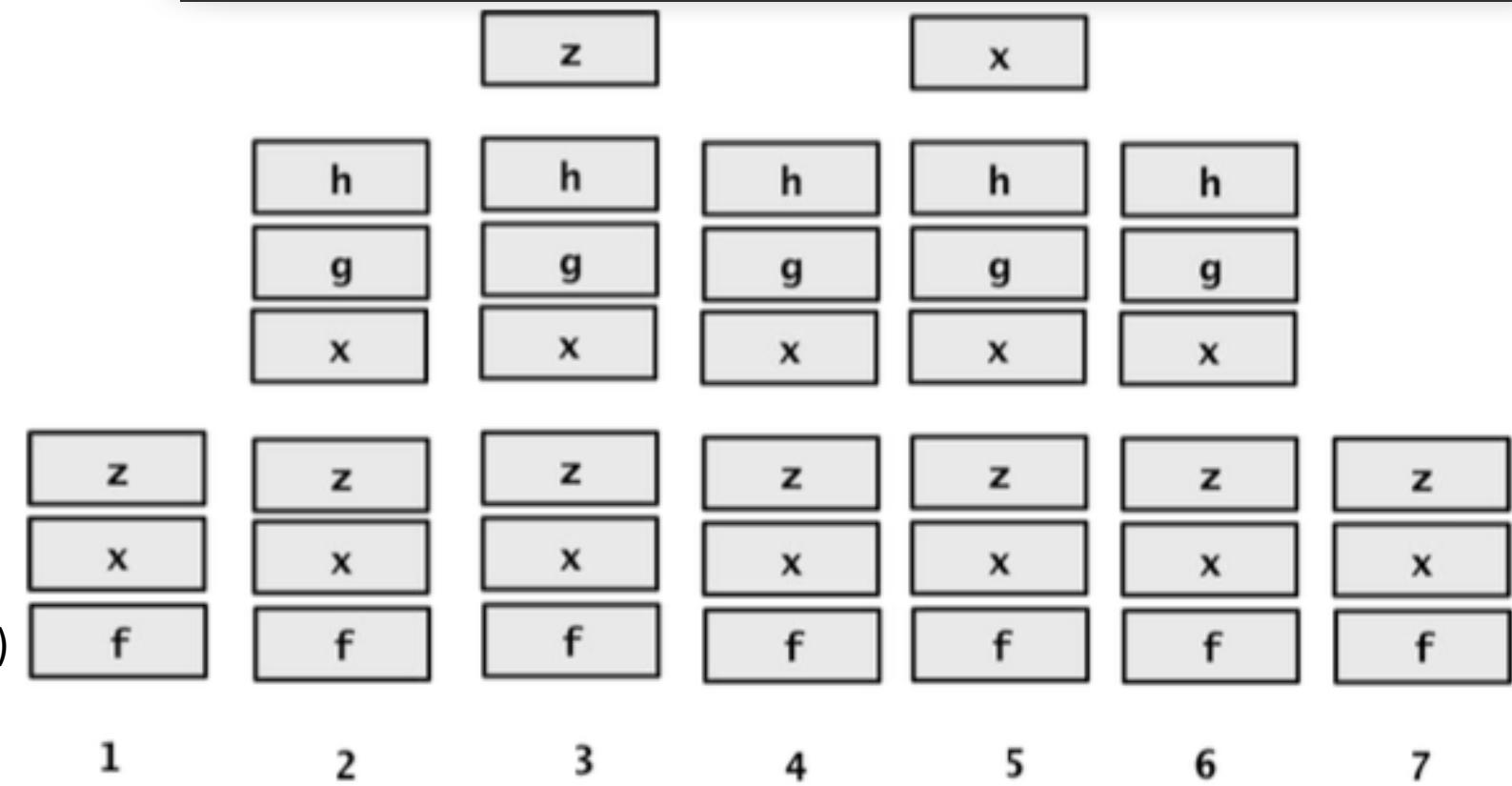
```

Output:

```

f1::x = 4
h::z = 4
g::x = abc
f1::x = 4
global::x = 3
global::z = <function f.<locals>.g at 0x7fb150904550>
g::x = abc

```



Reading Assignment

From Guttag's book, read Section 4.2 (Specifications)

Lecture 9

Recursion

Dr. Hacer Yalım Keleş

hkeles@ankara.edu.tr

Disclaimer: Content of this lecture slides are mostly from **Chapter 4** of Guttag's Book: "*Introduction to Computation and Programming Using Python*".

Principle

- Function calling itself

```
def func_f(x):  
    ...  
    func_f(x')
```

- Function, calling another function that calls this function back

```
def func_f(x):  
    ...  
    func_g(x')
```

```
def func_g(x):  
    ...  
    func_f(x')
```



Factorial Function

$$N! = N * N-1 * N-2 * \dots * 2 * 1$$

Factorial Function

$$N! = N * N-1 * N-2 * \dots * 2 * 1$$

$$N! = N * (N-1)!$$

Recursive definition

Factorial Function

$$N! = N * N-1 * N-2 * \dots * 2 * 1$$

$$N! = N * (N-1)!$$

Recursive definition

```
def factorial(x):  
    return x*factorial (x-1)
```

```
>>>factorial(3)
```

Factorial Function

$$N! = N * N-1 * N-2 * \dots * 2 * 1$$

$$N! = N * (N-1)!$$

Recursive definition

```
def factorial(x):  
    return x*factorial (x-1)
```

```
>>>factorial(3)
```



processing...

Factorial Function

$$N! = N * N-1 * N-2 * \dots * 2 * 1$$

$$N! = N * (N-1)!$$

Recursive definition

Consider including a base case

```
def factorial(x):  
    return x*factorial (x-1)
```

```
>>>factorial(3)
```



```
def factorial(x):  
    if x<=1:  
        return 1  
    return x*factorial (x-1)
```

base case of our inductive definition:



still processing...

Factorial Function

$$N! = N * N-1 * N-2 * \dots * 2 * 1$$

$$N! = N * (N-1)!$$

Recursive definition

Consider including a base case

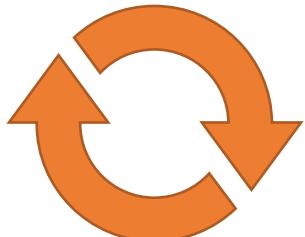
```
def factorial(x):  
    return x*factorial (x-1)
```



```
>>>factorial(3)
```

```
def factorial(x):  
    if x<=1:  
        return 1  
    return x*factorial (x-1)
```

```
>>>factorial(3)  
>>>6
```



yep, right! still processing...

Factorial Function

recursive definition

```
def factorial(x):  
    if x<=1:  
        return 1  
    return x*factorial (x-1)
```

```
>>>factorial(3)  
>>>6
```

iterative definition

```
def factorial(x):  
    res = 1  
    for i in range(1,x+1):  
        res *= i  
    return res
```

```
>>>factorial(3)  
>>>6
```

Factorial Function

recursive definition

```
def factorial(x):  
    if x<=1:  
        return 1  
    return x*factorial (x-1)
```

```
>>>factorial(3)  
>>>6
```

```
factorial(3)  
3*factorial(2)  
2*factorial(1)
```

iterative definition

```
def factorial(x):  
    res = 1  
    for i in range(1,x+1):  
        res *= i  
    return res
```

```
>>>factorial(3)  
>>>6
```

Factorial Function

recursive definition

```
def factorial(x):  
    if x<=1:  
        return 1  
    return x*factorial (x-1)
```

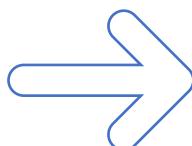
iterative definition

```
def factorial(x):  
    res = 1  
    for i in range(1,x+1):  
        res *= i  
    return res
```

```
>>>factorial(3)  
>>>6
```

```
>>>factorial(3)  
>>>6
```

```
factorial(3)  
3*factorial(2)  
2*factorial(1)
```



```
factorial(3)  
3*factorial(2)  
2*1
```

Factorial Function

recursive definition

```
def factorial(x):  
    if x<=1:  
        return 1  
    return x*factorial (x-1)
```

iterative definition

```
def factorial(x):  
    res = 1  
    for i in range(1,x+1):  
        res *= i  
    return res
```

```
>>>factorial(3)  
>>>6
```

```
>>>factorial(3)  
>>>6
```

```
factorial(3)  
3*factorial(2)  
2*factorial(1)
```



```
factorial(3)  
3*factorial(2)  
2*1
```



```
factorial(3)  
3*2*1
```

Factorial Function

recursive definition

```
def factorial(x):  
    if x<=1:  
        return 1  
    return x*factorial (x-1)
```

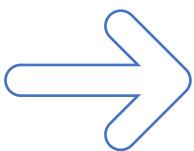
```
>>>factorial(3)  
>>>6
```

iterative definition

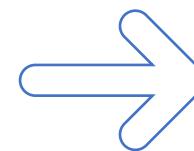
```
def factorial(x):  
    res = 1  
    for i in range(1,x+1):  
        res *= i  
    return res
```

```
>>>factorial(3)  
>>>6
```

```
factorial(3)  
3*factorial(2)  
2*factorial(1)
```



```
factorial(3)  
3*factorial(2)  
2*1
```



```
factorial(3)  
3*2*1
```



```
3*2*1
```



Fibonacci Function



https://www.youtube.com/watch?v=T8xgfVzef_E

`Fibonacci(0) = 1`

`Fibonacci(1) = 1`

`Fibonacci(N) = Fibonacci(N-1)+Fibonacci(N-2)`

Fibonacci Function



https://www.youtube.com/watch?v=T8xgfVzef_E

Can you write the recursive and iterative versions of this cool function?

`Fibonacci(0) = 1`

`Fibonacci(1) = 1`

`Fibonacci(N) = Fibonacci(N-1)+Fibonacci(N-2)`

Reading Assignment

From Guttag's book, read Section 4.3 (Recursion)

Lecture 10

Structured Types

Dr. Hacer Yalım Keleş

hkeles@ankara.edu.tr

Disclaimer: Content of this lecture slides are mostly from **Chapter 4** of Guttag's Book: "*Introduction to Computation and Programming Using Python*". Some slide contents are taken from https://python-textbook.readthedocs.io/en/1.0/Variables_and_Scope.html

Overview

- Tuples
- Lists
- Dictionaries

Tuples

- Ordered sequence of elements, like strings
- The elements can be of different types
- Immutable types,

We define a tuple *literal* by putting a comma-separated list of values inside round brackets (`(` and `)`):

Tuples

- Ordered sequence of elements, like strings
- The elements can be of different types
- Immutable types,

Tuple literals:

```
>>> t = () #empty tuple
```

```
t1 = (1, 'a', 2.9)
t2 = (t1, 'hello')
print(t1)
print(t2)
```

We define a tuple *literal* by putting a comma-separated list of values inside round brackets ((and)):

Tuples

- Ordered sequence of elements, like strings
- The elements can be of different types
- Immutable types,

We define a tuple *literal* by putting a comma-separated list of values inside round brackets ((and)):

Tuple literals:

```
>>> t = () #empty tuple
```

```
t1 = (1, 'a', 2.9)
t2 = (t1, 'hello')
print(t1)
print(t2)
```

```
(1, 'a', 2.9)
((1, 'a', 2.9), 'hello')
```

Tuples are objects; they can contain tuples

Tuples

```
t1 = (1, 'a', 2.9)
t2 = (t1, 'hello')
print(t1)
print(t2)
```

```
(1, 'a', 2.9)
((1, 'a', 2.9), 'hello')
```

```
print(t1+t2)
```

```
(1, 'a', 2.9, (1, 'a', 2.9), 'hello')
```

Tuples

```
t1 = (1, 'a', 2.9)
t2 = (t1, 'hello')
print(t1)
print(t2)
```

```
(1, 'a', 2.9)
((1, 'a', 2.9), 'hello')
```

```
print(t1+t2)
```

```
(1, 'a', 2.9, (1, 'a', 2.9), 'hello')
```

```
t = t1+t2
print(t[3])
```

```
(1, 'a', 2.9)
```

Tuples

```
t1 = (1, 'a', 2.9)
t2 = (t1, 'hello')
print(t1)
print(t2)
```

```
(1, 'a', 2.9)
((1, 'a', 2.9), 'hello')
```

```
print(t1+t2)
```

```
(1, 'a', 2.9, (1, 'a', 2.9), 'hello')
```

```
t = t1+t2
print(t[3])
```

```
(1, 'a', 2.9)
```

```
type(t[3])
```

```
tuple
```

Tuples

```
t1 = (1, 'a', 2.9)
t2 = (t1, 'hello')
print(t1)
print(t2)
```

```
(1, 'a', 2.9)
((1, 'a', 2.9), 'hello')
```

```
print(t1+t2)    Concatenation
```

```
(1, 'a', 2.9, (1, 'a', 2.9), 'hello')
```

```
t = t1+t2
print(t[3])
```

```
(1, 'a', 2.9)
```

```
type(t[3])
```

Indexing

```
tuple
```

```
print(t[2:5])
```

Slicing Generates new tuple

```
(2.9, (1, 'a', 2.9), 'hello')
```

Tuples

To create a singleton tuple, i.e. tuple literal with one element, a comma is necessary:

```
>>> t = (3,)
```

```
>>> t = (3) ← Simply int 3 within parenthesis
>>> print(t)
3 ←
>>> type(t)
<class 'int'>
```

```
<class 'int'>
>>> t = (3,) ← Comma is necessary to define singleton tuples
>>> type(t)
<class 'tuple'>
```

Tuples

```
def findDivisors (n1, n2):
    """Assumes that n1 and n2 are positive ints
    Returns a tuple containing all common divisors
    of n1 & n2"""
    divisors = () #the empty tuple
    for i in range(1, min (n1, n2) + 1):
        if n1%i == 0 and n2%i == 0:
            divisors = divisors + (i,)
    return divisors

divisors = findDivisors(20, 100)
print(divisors)
total = 0
for d in divisors:
    total += d
print("Total:",total)
```

write a function
findDivisors that computes
the common divisors of two
positive numbers

Tuples

```
def findDivisors (n1, n2):
    """Assumes that n1 and n2 are positive ints
    Returns a tuple containing all common divisors
    of n1 & n2"""
    divisors = () #the empty tuple
    for i in range(1, min (n1, n2) + 1):
        if n1%i == 0 and n2%i == 0:
            divisors = divisors + (i,)
    return divisors
```

```
divisors = findDivisors(20, 100)
print(divisors)
total = 0
for d in divisors:
    total += d
print("Total:",total)
```

write a function
findDivisors that computes
the common divisors of two
positive numbers

Using tuples, functions can return multiple values

Tuples

```
def findDivisors (n1, n2):
    """Assumes that n1 and n2 are positive ints
    Returns a tuple containing all common divisors
    of n1 & n2"""
    divisors = () #the empty tuple
    for i in range(1, min (n1, n2) + 1):
        if n1%i == 0 and n2%i == 0:
            divisors = divisors + (i,)
    return divisors

divisors = findDivisors(20, 100)
print(divisors)
total = 0
for d in divisors:
    total += d
print("Total:",total)
```

write a function
findDivisors that computes
the common divisors of two
positive numbers

Output:

(1, 2, 4, 5, 10, 20)
Total: 42

Tuples

if you know the length of a tuple or string, you can extract individual elements using multiple assignment

```
x, y = ("hello", 3)  
print(x)  
print(y)
```

```
hello  
3
```

Tuples

if you know the length of a tuple or string, you can extract individual elements using multiple assignment

```
x, y = ("hello", 3, 4)  
print(x)  
print(y)
```

```
-----  
ValueError                                     Traceback (most recent call last)  
<ipython-input-92-799ecde4a562> in <module>  
----> 1 x, y = ("hello", 3, 4)  
      2 print(x)  
      3 print(y)  
  
ValueError: too many values to unpack (expected 2)
```

Tuples

if you know the length of a tuple or **string**, you can extract individual elements using multiple assignment

```
a, b, c, d, e = "12345"  
print(d)  
print(type(d))
```

```
4  
<class 'str'>
```

Tuples

if you know the length of a tuple or string, you can extract individual elements using multiple assignment

This mechanism is particularly convenient when used in conjunction with functions that return fixed-size sequences.

This mechanism is particularly convenient when used in conjunction with functions that return fixed-size sequences.

```
def findExtremeDivisors(n1, n2):
    """Assumes that n1 and n2 are positive ints
    Returns a tuple containing the smallest common
    divisor > 1 and the largest common divisor of n1 and n2"""
    minVal, maxVal = None, None
    for i in range(2, min(n1, n2) + 1):
        if n1 % i == 0 and n2 % i == 0:
            if minVal == None or i < minVal:
                minVal = i
            if maxVal == None or i > maxVal:
                maxVal = i

    return (minVal, maxVal)
```

This mechanism is particularly convenient when used in conjunction with functions that return fixed-size sequences.

```
def findExtremeDivisors(n1, n2):
    """Assumes that n1 and n2 are positive ints
    Returns a tuple containing the smallest common
    divisor > 1 and the largest common divisor of n1 and n2"""
    minVal, maxVal = None, None
    for i in range(2, min(n1, n2) + 1):
        if n1 % i == 0 and n2 % i == 0:
            if minVal == None or i < minVal:
                minVal = i
            if maxVal == None or i > maxVal:
                maxVal = i

    return (minVal, maxVal)
```

short circuit evaluation

```
a = True
if a or b:
    print(a)
```

```
a = False
if a and b:
    print(a)
```

This mechanism is particularly convenient when used in conjunction with functions that return fixed-size sequences.

```
def findExtremeDivisors(n1, n2):
    """Assumes that n1 and n2 are positive ints
    Returns a tuple containing the smallest common
    divisor > 1 and the largest common divisor of n1 and n2"""
    minVal, maxVal = None, None
    for i in range(2, min(n1, n2) + 1):
        if n1 % i == 0 and n2 % i == 0:
            if minVal == None or i < minVal:
                minVal = i
            if maxVal == None or i > maxVal:
                maxVal = i
    return (minVal, maxVal)

minDivisor, maxDivisor = findExtremeDivisors(100, 200)
print(minDivisor)
print(maxDivisor)
```

2
100

Tuples

We can

```
animals = ('cat', 'bird', 'dog', 'bird')
count = animals.count('bird')
idx = animals.index('dog')
print("count of bird:", count)
print("index of dog:", idx)
```

```
count of bird: 2
index of dog: 2
```

Tuples

```
animals = ('cat', 'bird', 'dog', 'bird')
count = animals.count('bird')
idx = animals.index('dog')
print("count of bird:", count)
print("index of dog:", idx)
```

```
count of bird: 2
index of dog: 2
```

We can NOT

```
animals.append('tiger')
animals[1] = 'crocodile'
```

What are tuples good for?

We can use them to create a sequence of values that we don't want to modify.
For example, the list of weekday names is never going to change.

What are tuples good for?

We can use them to create formatted strings

```
s = "my name is %s, I have %d books, my GPA is %f" % ("John", 15, 3.96)
print(s)
```

```
my name is John, I have 15 books, my GPA is 3.960000
```

Lecture 11

Structured Types - Lists

Dr. Hacer Yalım Keleş

hkeles@ankara.edu.tr

Disclaimer: Content of this lecture slides are mostly from **Chapter 5** of Guttag's Book: "*Introduction to Computation and Programming Using Python*". Some slide contents are taken from <https://python-textbook.readthedocs.io/en/1.0>

Overview

- Tuples
- Lists
- Dictionaries

Lists

- Ordered sequence of elements, like tuples,
- Indexes are used to access the elements
- Singleton lists can be created without comma at the end :)
- Mutable types

We define a list *literal* by putting a comma-separated list of values inside square brackets (`[` and `]`):

Lists

- Ordered sequence of elements, like tuples,
- Indexes are used to access the elements
- Singleton lists can be created without comma at the end :)
- Mutable types

```
animals = ['cat', 'dog', 'bird', 'fish']
print(animals[2:4])
print(animals[1])
print(animals[-1:])
print(animals[:-1])
```

We define a list *literal* by putting a comma-separated list of values inside square brackets ([and]):

Lists

- Ordered sequence of elements, like tuples,
- Indexes are used to access the elements
- Singleton lists can be created without comma at the end :)
- Mutable types

```
animals = ['cat', 'dog', 'bird', 'fish']
print(animals[2:4])
print(animals[1])
print(animals[-1:])
print(animals[:-1])
```

```
['bird', 'fish']
dog
['fish']
['cat', 'dog', 'bird']
```

We define a list *literal* by putting a comma-separated list of values inside square brackets ([and]):

Lists

- Ordered sequence of elements, like tuples,
- Indexes are used to access the elements
- Singleton lists can be created without comma at the end :)
- Mutable types

We define a list *literal* by putting a comma-separated list of values inside square brackets ([and]):

```
# we can concatenate two lists by adding them
print([1, 2, 3] + [4, 5, 6])

# we can concatenate a list with itself by multiplying it by an integer
print([1, 2, 3] * 3)
```

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

Lists

- Ordered sequence of elements, like tuples,
- Indexes are used to access the elements
- Singleton lists can be created without comma at the end :)
- Mutable types

```
x = ["hello", 2, 3.46, "Comic Books"]
for i in x:
    print(i, end=' ')
```

```
hello 2 3.46 Comic Books
```

We define a list *literal* by putting a comma-separated list of values inside square brackets ([and]):

Lists

- Ordered sequence of elements, like tuples,
- Indexes are used to access the elements
- Singleton lists can be created without comma at the end :)
- Mutable types

```
x = ["hello", 2, 3.46, "Comic Books"]
for i in x:
    print(i, end=' ')
```

```
hello 2 3.46 Comic Books
```

```
a = x[3][6:][3]
print(a)
```



We define a list *literal* by putting a comma-separated list of values inside square brackets ([and]):

Lists

- Ordered sequence of elements, like tuples,
- Indexes are used to access the elements
- Singleton lists can be created without comma at the end :)
- Mutable types

```
x = ["hello", 2, 3.46, "Comic Books"]
for i in x:
    print(i, end=' ')
```

```
hello 2 3.46 Comic Books
```

```
a = x[3][6:][3]
print(a)
```



We define a list *literal* by putting a comma-separated list of values inside square brackets ([and]):

Lists

- Ordered sequence of elements, like tuples,
- Indexes are used to access the elements
- Singleton lists can be created without comma at the end :)
- Mutable types

We define a list *literal* by putting a comma-separated list of values inside square brackets ([and]):

```
L1 = [1,2,3]
L2 = [1,2,3]
if L1==L2:
    print("same contents")
if L1 is L2:
    print("same objects")
```

Operator `is` is used to check the equivalence of objects

Lists

- Ordered sequence of elements, like tuples,
- Indexes are used to access the elements
- Singleton lists can be created without comma at the end :)
- Mutable types

We define a list *literal* by putting a comma-separated list of values inside square brackets ([and]):

```
L1 = [1,2,3]
L2 = [1,2,3]
if L1==L2:
    print("same contents")
if L1 is L2:
    print("same objects")
```

Operator `is` is used to check the equivalence of objects

Output: same contents

Lists

- Ordered sequence of elements, like tuples,
- Indexes are used to access the elements
- Singleton lists can be created without comma at the end :)
- Mutable types

```
animals = ['cat', 'dog', 'bird', 'fish']
pets = animals
print(id(animals))
print(id(pets))
```

```
140352227448128
140352227448128
```

Function **id(obj)** is used to get the id of object **obj**. Different objects have distinct ids.

Both of the variables, animals and pets, refer to the same list object (aliasing)

in and not in operators with lists

```
L1 = ["john", "elisa", "mat", "geo", "com"]
s = "mat"
if s in L1:
    print("yes")

if "matte" not in L1:
    print("no")
```

in and not in operators with lists

```
L1 = ["john", "elisa", "mat", "geo", "com"]
s = "mat"
if s in L1:
    print("yes")

if "matte" not in L1:
    print("no")
```

```
yes
no
```

Value vs content comparisons

```
a = [1,2]
b = ['hello','bye']
c = [a, b]
d = [a, b]
print(c)
print(d)
print("id of c:",id(c))
print("id of d:",id(d))
print("id of c[0]:",id(c[0]))
print("id of d[0]:",id(d[0]))
print(c==d) # value (content) check
print(id(c)==id(d)) # object check
```

Value vs content comparisons

```
a = [1,2]
b = ['hello', 'bye']
c = [a, b]
d = [a, b]
print(c)
print(d)
print("id of c:", id(c))
print("id of d:", id(d))
print("id of c[0]:", id(c[0]))
print("id of d[0]:", id(d[0]))
print(c==d) # value (content) check
print(id(c)==id(d)) # object check
```

Output:

```
[[1, 2], ['hello', 'bye']]
[[1, 2], ['hello', 'bye']]
id of c: 140607650863360
id of d: 140607650864384
id of c[0]: 140607650863104
id of d[0]: 140607650863104
True
False
```

aliasing: there are two distinct paths to list objects a and b; through objects c and d...

Aliasing can create undesired side effects..be careful!

```
a = [1,2]
b = ['hello','bye']
c = [a, b]
d = [a, b]
c[0][0] = 4
print(c)
print(d)
print(id(c)==id(d))
print(id(c[0])==id(d[0]))
```

Aliasing can create undesired side effects..be careful!

```
a = [1, 2]
b = ['hello', 'bye']
c = [a, b]
d = [a, b]
c[0][0] = 4
print(c)
print(d)
print(id(c)==id(d))
print(id(c[0])==id(d[0]))
```

Output:

```
[[4, 2], ['hello', 'bye']]
[[4, 2], ['hello', 'bye']]
False
True
```

Aliasing can create undesired side effects..be careful!

```
a = [1,2]
b = ['hello','bye']
c = [a, b]
d = [a, b]
c[1][0] = 4
print(c)
print(d)
```

```
[[4, 2], [4, 'bye']]
[[1, 2], [4, 'bye']]
```

Aliasing can create undesired side effects..be careful!

```
t =()
print(id(t))
t +=('3',)
print(id(t))
t+=(2,3,4)
print(id(t))
print(t)
```

```
140352095891520
140352228002208
140352227995296
('3', 2, 3, 4)
```

```
t=([2,3,4],5)
print(id(t))
print(t)
```

```
140352227049280
([2, 3, 4], 5)
```

Lists are mutable

Even within the tuple context

```
print(id(t))
t[0][0] = 'a'
print(t)
print(id(t))
```

```
140352227049280
(['a', 3, 4], 5)
140352227049280
```

List operations

```
L1 = [ 'a' , 'b' , 'c' ]
L2 = [ 1 , 2 , 3 ]
L3 = L1+L2
print(L1,L2,L3,sep=' \n ')
```

List operations

```
L1 = ['a', 'b', 'c']
L2 = [1, 2, 3]
L3 = L1+L2
print(L1,L2,L3,sep='\n')
```

```
['a', 'b', 'c']
[1, 2, 3]
['a', 'b', 'c', 1, 2, 3]
```

List operations

```
L1 = ['a', 'b', 'c']
L2 = [1, 2, 3]
→ L2.extend(L1)
L3 = L1+L2
print(L1,L2,L3,sep=' \n ')
```

```
['a', 'b', 'c']
[1, 2, 3, 'a', 'b', 'c']
['a', 'b', 'c', 1, 2, 3, 'a', 'b', 'c']
```

List operations

```
L1 = ['a', 'b', 'c']
L2 = [1, 2, 3]
→ L1.append(L2)
print(L1,L2,sep='\n')
```

```
['a', 'b', 'c', [1, 2, 3]]
[1, 2, 3]
```

List operations

`L.append(e)` adds the object `e` to the end of `L`.

`L.count(e)` returns the number of times that `e` occurs in `L`.

`L.insert(i, e)` inserts the object `e` into `L` at index `i`.

`L.extend(L1)` adds the items in list `L1` to the end of `L`.

`L.remove(e)` deletes the first occurrence of `e` from `L`.

`L.index(e)` returns the index of the first occurrence of `e` in `L`. It raises an exception (see Chapter 7) if `e` is not in `L`.

`L.pop(i)` removes and returns the item at index `i` in `L`. If `i` is omitted, it defaults to `-1`, to remove and return the last element of `L`.

`L.sort()` sorts the elements of `L` in ascending order.

`L.reverse()` reverses the order of the elements in `L`.

Note that, all of these operations, except for the count and index, mutate the list object.

List operations

`L.append(e)` adds the object e to the end of L.

`L.count(e)` returns the number of times that e occurs in L.

`L.insert(i, e)` inserts the object e into L at index i.

`L.extend(L1)` adds the items in list L1 to the end of L.

`L.remove(e)` deletes the first occurrence of e from L.

`L.index(e)` returns the index of the first occurrence of e in L. It raises an exception (see Chapter 7) if e is not in L.

`L.pop(i)` removes and returns the item at index i in L. If i is omitted, it defaults to -1, to remove and return the last element of L.

`L.sort()` sorts the elements of L in ascending order.

`L.reverse()` reverses the order of the elements in L.

Note that, all of these operations, except for the count and index, mutate the list object.

```
L1 = ['a', 'b', 'c']
L2 = [1, 2, 3]
L1.remove('b')
L2.pop(2)
print(L1,L2,sep='\n')
```

```
['a', 'c']
[1, 2]
```

List operations

`L.append(e)` adds the object `e` to the end of `L`.

`L.count(e)` returns the number of times that `e` occurs in `L`.

`L.insert(i, e)` inserts the object `e` into `L` at index `i`.

`L.extend(L1)` adds the items in list `L1` to the end of `L`.

`L.remove(e)` deletes the first occurrence of `e` from `L`.

`L.index(e)` returns the index of the first occurrence of `e` in `L`. It raises an exception (see Chapter 7) if `e` is not in `L`.

`L.pop(i)` removes and returns the item at index `i` in `L`. If `i` is omitted, it defaults to `-1`, to remove and return the last element of `L`.

`L.sort()` sorts the elements of `L` in ascending order.

`L.reverse()` reverses the order of the elements in `L`.

Note that, all of these operations, except for the count and index, mutate the list object.

```
L1 = ['a', 'b', 'c']
L2 = [1, 2, 3]
L1.insert(1,L2)
print(L1,L2,sep='\n')
L2[2] = '?'
print(L1,L2,sep='\n')
```

List operations

`L.append(e)` adds the object `e` to the end of `L`.

`L.count(e)` returns the number of times that `e` occurs in `L`.

`L.insert(i, e)` inserts the object `e` into `L` at index `i`.

`L.extend(L1)` adds the items in list `L1` to the end of `L`.

`L.remove(e)` deletes the first occurrence of `e` from `L`.

`L.index(e)` returns the index of the first occurrence of `e` in `L`. It raises an exception (see Chapter 7) if `e` is not in `L`.

`L.pop(i)` removes and returns the item at index `i` in `L`. If `i` is omitted, it defaults to `-1`, to remove and return the last element of `L`.

`L.sort()` sorts the elements of `L` in ascending order.

`L.reverse()` reverses the order of the elements in `L`.

Note that, all of these operations, except for the count and index, mutate the list object.

```
L1 = ['a', 'b', 'c']
L2 = [1, 2, 3]
L1.insert(1,L2)
print(L1,L2,sep='\n')
L2[2] = '?'
print(L1,L2,sep='\n')
```

```
['a', [1, 2, 3], 'b', 'c']
[1, 2, 3]
['a', [1, 2, '?'], 'b', 'c']
[1, 2, '?']
```

Aliasing

List operations

`L.append(e)` adds the object `e` to the end of `L`.

`L.count(e)` returns the number of times that `e` occurs in `L`.

`L.insert(i, e)` inserts the object `e` into `L` at index `i`.

`L.extend(L1)` adds the items in list `L1` to the end of `L`.

`L.remove(e)` deletes the first occurrence of `e` from `L`.

`L.index(e)` returns the index of the first occurrence of `e` in `L`. It raises an exception (see Chapter 7) if `e` is not in `L`.

`L.pop(i)` removes and returns the item at index `i` in `L`. If `i` is omitted, it defaults to `-1`, to remove and return the last element of `L`.

`L.sort()` sorts the elements of `L` in ascending order.

`L.reverse()` reverses the order of the elements in `L`.

Note that, all of these operations, except for the count and index, mutate the list object.

```
L1 = ['c', 'b', 'a', 'k']
L2 = [41, 12, 3, 60, 15]
L2.sort()
L1.reverse()
print(L1,L2,sep='\n')
```

```
['k', 'a', 'b', 'c']
[3, 12, 15, 41, 60]
```

List operations

`L.append(e)` adds the object `e` to the end of `L`.

`L.count(e)` returns the number of times that `e` occurs in `L`.

`L.insert(i, e)` inserts the object `e` into `L` at index `i`.

`L.extend(L1)` adds the items in list `L1` to the end of `L`.

`L.remove(e)` deletes the first occurrence of `e` from `L`.

`L.index(e)` returns the index of the first occurrence of `e` in `L`. It raises an exception (see Chapter 7) if `e` is not in `L`.

`L.pop(i)` removes and returns the item at index `i` in `L`. If `i` is omitted, it defaults to `-1`, to remove and return the last element of `L`.

`L.sort()` sorts the elements of `L` in ascending order.

`L.reverse()` reverses the order of the elements in `L`.

Note that, all of these operations, except for the count and index, mutate the list object.

```
L1 = [1, 2, 3]
```

```
L2 = L1[ : ]
```

```
print(id(L1))
```

```
print(id(L2))
```

```
140352227456128
```

```
140352227963072
```

Slicing returns a new list object

List operations

`L.append(e)` adds the object `e` to the end of `L`.

`L.count(e)` returns the number of times that `e` occurs in `L`.

`L.insert(i, e)` inserts the object `e` into `L` at index `i`.

`L.extend(L1)` adds the items in list `L1` to the end of `L`.

`L.remove(e)` deletes the first occurrence of `e` from `L`.

`L.index(e)` returns the index of the first occurrence of `e` in `L`. It raises an exception (see Chapter 7) if `e` is not in `L`.

`L.pop(i)` removes and returns the item at index `i` in `L`. If `i` is omitted, it defaults to `-1`, to remove and return the last element of `L`.

`L.sort()` sorts the elements of `L` in ascending order.

`L.reverse()` reverses the order of the elements in `L`.

Note that, all of these operations, except for the count and index, mutate the list object.

```
animals = ['cat', 'dog', 'goldfish', 'canary']
pets = list(animals)

animals.sort()
pets.append('gerbil')

print(animals)
print(pets)
```

```
['canary', 'cat', 'dog', 'goldfish']
['cat', 'dog', 'goldfish', 'canary', 'gerbil']
```

`list(animals)` create a new list object, copying the content from `animals`.

Built-in functions and lists

```
animals = ['cat', 'dog', 'goldfish', 'canary']
numbers = [1, 2, 3, 4, 5,] #trailing comma is ok in python

# the length of a list
print(len(animals))

# the sum of a list of numbers
print(sum(numbers))

# are any of these values true?
if any([1,0,1,0,1]):
    print("yes, some values are true")

# are all of these values true?
if not all([1,0,1,0,1]):
    print("yep, not all is true")
```

Copying lists

```
def removeDuplicates(L1, L2):
    """Assumes that L1 and L2 are lists.
    Removes any element from L1 that also
    occurs in L2"""
    for e1 in L1:
        if e1 in L2:
            L1.remove(e1)
L1 = [1,2,3,4]
L2 = [1,2,5,6]
removeDuplicates(L1, L2)
print("L1:",L1)
```

Copying lists

```
def removeDuplicates(L1, L2):
    """Assumes that L1 and L2 are lists.
    Removes any element from L1 that also
    occurs in L2"""
    for e1 in L1:
        if e1 in L2:
            L1.remove(e1)
L1 = [1,2,3,4]
L2 = [1,2,5,6]
removeDuplicates(L1, L2)
print("L1:",L1)
```

L1: [2, 3, 4]

What is wrong here?

We are modifying the list during iterations,
list is mutated and items' positions change;
which effect the algorithmic integrity.

We should get a copy of L1 and iterate over that copy

Copying lists

```
def removeDuplicates(L1, L2):
    """Assumes that L1 and L2 are lists.
    Removes any element from L1 that also
    occurs in L2"""
    for el in L1[:]:           Slicing creates a copy of L1
        if el in L2:
            L1.remove(el)

L1 = [1,2,3,4]
L2 = [1,2,5,6]
removeDuplicates(L1, L2)
print("L1:",L1)
```

L1: [3, 4]

Copying lists

```
def removeDuplicates(L1, L2):
    """Assumes that L1 and L2 are lists.
    Removes any element from L1 that also
    occurs in L2"""
    for el in list(L1):    list(L1) created a new object copying from L1
        if el in L2:
            L1.remove(el)
L1 = [1,2,3,4]
L2 = [1,2,5,6]
removeDuplicates(L1, L2)
print("L1:",L1)
```

```
L1: [3, 4]
```

Copying lists

```
import copy

def removeDuplicates(L1, L2):
    """Assumes that L1 and L2 are lists.
    Removes any element from L1 that also
    occurs in L2"""
    Ltemp = copy.deepcopy(L1)
    for e1 in Ltemp:
        if e1 in L2:
            L1.remove(e1)

L1 = [1,2,3,4]
L2 = [1,2,5,6]
removeDuplicates(L1, L2)
print("L1:", L1)
```

copy.deepcopy creates a new object copying from L1.
if L1 contained mutable objects inside, they would also
be copied (no aliasing).

```
L1: [3, 4]
```

Passing lists as arguments to functions

```
def f(L):
    newL = L
    for i in range(len(newL)):
        newL[i] = i
    return newL

mylist = [1.2,2.5,3.0]
a = f(mylist)
print(a)
print(mylist)
```

Passing lists as arguments to functions

```
def f(L):
    newL = L
    for i in range(len(newL)):
        newL[i] = i
    return newL

mylist = [1.2,2.5,3.0]
a = f(mylist)
print(a)
print(mylist)
```

```
[0, 1, 2]
[0, 1, 2]
```

List comprehension

List comprehension provides a concise way to apply an operation to the values in a sequence.

```
L = [x**2 for x in range(1,7)]  
print(L)
```

```
[1, 4, 9, 16, 25, 36]
```

List comprehension

List comprehension provides a concise way to apply an operation to the values in a sequence.

```
L = [x**2 for x in range(1,7)]  
print(L)
```

```
[1, 4, 9, 16, 25, 36]
```

```
L = [x**2 for x in range(1,7) if x%2==0]  
print(L)
```

```
[4, 16, 36]
```

```
L = [x**2 for x in range(1,7) if x%2==0 and x>4]  
print(L)
```

```
[36]
```

Functions and lists

In python, functions are first class objects.

They can be treated like any other objects, i.e. int, list etc.

using functions as arguments can be particularly convenient in conjunction with lists.

It allows a style of coding called higher-order programming.

Functions and lists

In python, functions are first class objects.

They can be treated like any other objects, i.e. int, list etc.

using functions as arguments can be particularly convenient in conjunction with lists.

It allows a style of coding called higher-order programming.

```
def applyToEach(L, f):
    """Assumes L is a list, f a function
    Mutates L by replacing each element, e,
    of L by f(e)"""
    for i in range(len(L)):
        L[i] = f(L[i])
```

The function applyToEach is called higher-order because it has an argument that is itself a function.

Functions and lists

```
def applyToEach(L, f):
    """Assumes L is a list, f a function
    Mutates L by replacing each element, e,
    of L by f(e)"""
    for i in range(len(L)):
        L[i] = f(L[i])

L = [1, -2, -3.33, 4e-2]
print('L =', L)
print('Apply abs to each element of ', L)
applyToEach(L, abs)
print('L =', L)
print('Apply int to each element of ', L)
applyToEach(L, int)
print('L =', L)
print('Apply factorial to each element of ', L)
applyToEach(L, fact)
print('L =', L)
print('Apply Fibonnaci to each element of ', L)
applyToEach(L, fib)
print('L =', L)
```

Functions and lists

```
def applyToEach(L, f):
    """Assumes L is a list, f a function
    Mutates L by replacing each element, e,
    of L by f(e)"""
    for i in range(len(L)):
        L[i] = f(L[i])

L = [1, -2, -3.33, 4e-2]
print('L =', L)
print('Apply abs to each element of ', L)
applyToEach(L, abs)
print('L =', L)
print('Apply int to each element of ', L)
applyToEach(L, int)
print('L =', L)
print('Apply factorial to each element of ', L)
applyToEach(L, fact)
print('L =', L)
print('Apply Fibonnaci to each element of ', L)
applyToEach(L, fib)
print('L =', L)
```

```
L = [1, -2, -3.33, 0.04]
Apply abs to each element of [1, -2, -3.33, 0.04]
L = [1, 2, 3.33, 0.04]
Apply int to each element of [1, 2, 3.33, 0.04]
L = [1, 2, 3, 0]
Apply factorial to each element of [1, 2, 3, 0]
L = [1, 2, 6, 1]
Apply Fibonnaci to each element of [1, 2, 6, 1]
L = [1, 2, 13, 1]
```

Functions and lists

Python has a built-in higher-order function, `map`, that is similar to, but more general than, the `applyToEach` function

`map(fact, [1, 2, 3]) → [1, 2, 6].`

First argument is a unary function

Second argument is the sequence of arguments to the unary fact function

Functions and lists

Python has a built-in higher-order function, `map`, that is similar to, but more general than, the `applyToEach` function

```
map(fact, [1, 2, 3]) → [1, 2, 6].
```

More generally, the first argument to `map` can be of function of n arguments, in which case it must be followed by n subsequent ordered collections.

```
L1 = [1, 28, 36]  
L2 = [2, 57, 9]
```

```
map(min, L1, L2) → [1, 28, 9]
```

First argument, `min`, is a function that takes two arguments

Second and third arguments correspond to the sequences of args. for the `min` function.

Lecture 12

Structured Types - Dictionaries

Dr. Hacer Yalım Keleş

hkeles@ankara.edu.tr

Disclaimer: Content of this lecture slides are mostly from **Chapter 5** of Guttag's Book: "*Introduction to Computation and Programming Using Python*". Some slide contents are taken from <https://python-textbook.readthedocs.io/en/1.0>

Overview

- Tuples
- Lists
- Dictionaries

Dictionaries

- Unordered sequence of elements with key-value pairs
- Since unordered, instead of indexes we use keys to access the elements
- Dict literals can be created using curly braces, i.e. { }
- Mutable types

Dictionaries

- Unordered sequence of elements with key-value pairs
- Since unordered, instead of indexes we use keys to access the elements
- Dict literals can be created using curly braces, i.e. { }
- Mutable types

```
monthNumbers = {'Jan':1, 'Feb':2, 'Mar':3, 'Apr':4, 'May':5,  
                1:'Jan', 2:'Feb', 3:'Mar', 4:'Apr', 5:'May'}
```

Dictionaries

- Unordered sequence of elements with key-value pairs
- Since unordered, instead of indexes we use keys to access the elements
- Dict literals can be created using curly braces, i.e. { }
- Mutable types

```
monthNumbers = {'Jan':1, 'Feb':2, 'Mar':3, 'Apr':4, 'May':5,
                1:'Jan', 2:'Feb', 3:'Mar', 4:'Apr', 5:'May'}
```

```
student_info = {
    "id": 120,
    "name": "Jane",
    "surname": "Snow"
}
```

key: any immutable type

Dictionaries

- Unordered sequence of elements with key-value pairs
- Since unordered, instead of indexes we use keys to access the elements
- Dict literals can be created using curly braces, i.e. { }
- Mutable types

```
monthNumbers = {'Jan':1, 'Feb':2, 'Mar':3, 'Apr':4, 'May':5,  
                1:'Jan', 2:'Feb', 3:'Mar', 4:'Apr', 5:'May'}
```

```
student_info = {  
    "id": 120,  
    "name": "Jane",  
    "surname": "Snow"  
}
```

key: any immutable type

We can mix different type of keys and different type of values in one dictionary.

Dictionaries

```
monthNumbers = {'Jan':1, 'Feb':2, 'Mar':3, 'Apr':4, 'May':5,  
                1:'Jan', 2:'Feb', 3:'Mar', 4:'Apr', 5:'May'}  
  
print('The third month is ' + monthNumbers[3])  
dist = monthNumbers['Apr'] - monthNumbers['Jan']  
print('Apr and Jan are', dist, 'months apart')
```

Dictionaries

```
monthNumbers = {'Jan':1, 'Feb':2, 'Mar':3, 'Apr':4, 'May':5,  
                1:'Jan', 2:'Feb', 3:'Mar', 4:'Apr', 5:'May'}  
  
print('The third month is ' + monthNumbers[3])  
dist = monthNumbers['Apr'] - monthNumbers['Jan']  
print('Apr and Jan are', dist, 'months apart')
```

The third month is Mar

Apr and Jan are 3 months apart

Dictionaries: key() method

```
monthNumbers = {'Jan':1, 'Feb':2, 'Mar':3, 'Apr':4, 'May':5,  
                1:'Jan', 2:'Feb', 3:'Mar', 4:'Apr', 5:'May'}
```

The method `keys` returns a list containing the keys of a dictionary.
The order in which the keys appear is not defined.

```
print(monthNumbers.keys())
```

```
dict_keys(['Jan', 'Feb', 'Mar', 'Apr', 'May', 1, 2, 3, 4, 5])
```

Dictionaries: mutable types

```
monthNumbers = {'Jan':1, 'Feb':2, 'Mar':3, 'Apr':4, 'May':5,  
                1:'Jan', 2:'Feb', 3:'Mar', 4:'Apr', 5:'May'}
```

We add elements to a dictionary by assigning a value to an unused key

```
monthNumbers['June'] = 6  
monthNumbers[6] = 'June'
```

```
print(monthNumbers.keys())  
print(monthNumbers.values())
```

We use values() method to list the values of a dict.

```
dict_keys(['Jan', 'Feb', 'Mar', 'Apr', 'May', 1, 2, 3, 4, 5, 'June', 6])  
dict_values([1, 2, 3, 4, 5, 'Jan', 'Feb', 'Mar', 'Apr', 'May', 6, 'June'])
```

Dictionaries: mutable types

```
monthNumbers = { 'Jan':1, 'Feb':2, 'Mar':3, 'Apr':4, 'May':5,  
                 1:'Jan', 2:'Feb', 3:'Mar', 4:'Apr', 5:'May' }
```

```
monthNumbers['June'] = 6  
monthNumbers[6] = 'June'
```

We can update a dictionary object by inserting multiple items using update() method

```
monthNumbers.update({ 'July':7, 7:'July', 'Aug':8, 8:'Aug' })  
print(monthNumbers.keys())
```

Dictionaries: mutable types

```
monthNumbers = { 'Jan':1, 'Feb':2, 'Mar':3, 'Apr':4, 'May':5,  
                 1:'Jan', 2:'Feb', 3:'Mar', 4:'Apr', 5:'May' }
```

```
monthNumbers['June'] = 6  
monthNumbers[6] = 'June'
```

We can update a dictionary object by inserting multiple items using update() method

```
monthNumbers.update({ 'July':7, 7:'July', 'Aug':8, 8:'Aug' })  
print(monthNumbers.keys())
```

```
dict_keys(['Jan', 'Feb', 'Mar', 'Apr', 'May', 1, 2, 3, 4, 5, 'June', 6, 'July', 7, 'Aug', 8])
```

Dictionaries: Useful operations

len(d) returns the number of items in d.

d.keys() returns a list containing the keys in d.

d.values() returns a list containing the values in d.

k in d returns True if key k is in d.

d[k] returns the item in d with key k.

d.get(k, v) returns d[k] if k is in d, and v otherwise.

d[k] = v associates the value v with the key k in d. If there is already a value associated with k, that value is replaced.

del d[k] removes the key k from d.

for k in d iterates over the keys in d.

Example

```
student_info = {  
    "id": 120,  
    "name": "Jane",  
    "surname" : "Snow"  
}  
  
print(student_info.items())  
print("(",student_info["id"],")",student_info["name"], student_info["surname"])  
#update an item value using key  
student_info["name"] = "John"  
#update multiple item-value pairs  
student_info.update({"adress":"Ankara", "id": 176, "courses":"com1001"})  
print(student_info.keys())  
print(student_info.values())  
print(student_info.items())
```

Example

```
student_info = {  
    "id": 120,  
    "name": "Jane",  
    "surname" : "Snow"  
}  
  
print(student_info.items())  
print("(",student_info["id"],")",student_info["name"], student_info["surname"])  
#update an item value using key  
student_info[ "name" ] = "John"  
#update multiple item-value pairs  
student_info.update({ "adress": "Ankara", "id": 176, "courses": "com1001" })  
print(student_info.keys())  
print(student_info.values())  
print(student_info.items())
```

```
dict_items([('id', 120), ('name', 'Jane'), ('surname', 'Snow')])  
( 120 ) Jane Snow  
dict_keys(['id', 'name', 'surname', 'adress', 'courses'])  
dict_values([176, 'John', 'Snow', 'Ankara', 'com1001'])  
dict_items([('id', 176), ('name', 'John'), ('surname', 'Snow'), ('adress', 'Ankara'), ('courses', 'com1001')])
```

Example

```
student_info = {
    "id": 120,
    "name": "Jane",
    "surname" : "Snow"
}

print(student_info.items())
print("(",student_info["id"],")",student_info["name"], student_info["surname"])
#update an item value using key
student_info["name"] = "John"
#update multiple item-value pairs
student_info.update({"adress":"Ankara", "id": 176, "courses":"com1001"})
print(student_info.keys())
print(student_info.values())
print(student_info.items())
```

```
for i in student_info.keys():
    print(i,end=" ")
print() #by default prints newline

for i in student_info.values():
    print(i, end=" ")
print() #by default prints newline

for i,j in student_info.items():
    if i=="name" or i=="surname":
        print(j)
```

```
id name surname adress courses
176 John Snow Ankara com1001
John
Snow
```

Reading Assignment: Read Chapter 6 (Testing and Debugging) of Guttag's Book

Lecture 13

Exceptions

Dr. Hacer Yalım Keleş

hkeles@ankara.edu.tr

Disclaimer: Content of this lecture slides are mostly from **Chapter 7** of Guttag's Book: "*Introduction to Computation and Programming Using Python*".

Exception

Something that is not normal

Exception

Something that is not normal

There is nothing rare about **exceptions** in Python. They are everywhere.

Virtually every module in the standard Python library uses them, and Python itself will raise them in many different circumstances.

Exception

Something that is not normal

There is nothing rare about **exceptions** in Python. They are everywhere.

Virtually every module in the standard Python library uses them, and Python itself will raise them in many different circumstances.

```
>>> a = [1, 2, 3]
  File "<stdin>", line 1
    a = [1, 2, 3]
      ^
IndentationError: unexpected indent
```

Exception

Something that is not normal

There is nothing rare about **exceptions** in Python. They are everywhere.

Virtually every module in the standard Python library uses them, and Python itself will raise them in many different circumstances.

```
>>>→a = [1, 2, 3]
  File "<stdin>", line 1
    a = [1, 2, 3]
    ^
IndentationError: unexpected indent
```

Exception IndentationError raised by Python
When you write code with wrong indentation

```
>>> a = [1, 2, 3]
>>> a[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Exception IndexError raised by Python
When you exceed the bounds of list or similar objects

Exception

Something that is not normal

There is nothing rare about **exceptions** in Python. They are everywhere.

Virtually every module in the standard Python library uses them, and Python itself will raise them in many different circumstances.

```
>>>→a = [1, 2, 3]
  File "<stdin>", line 1
    a = [1, 2, 3]
    ^
IndentationError: unexpected indent
```

Exception IndentationError raised by Python
When you write code with wrong indentation

```
>>> a = [1, 2, 3]
>>> a[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Exception IndexError raised by Python
When you exceed the bounds of list or similar objects

Exception

Something that is not normal

There is nothing rare about **exceptions** in Python. They are everywhere.

Virtually every module in the standard Python library uses them, and Python itself will raise them in many different circumstances.

Among the most commonly occurring types of exceptions are `TypeError`, `NameError`, and `ValueError`

Exception

Something that is not normal

There is nothing rare about **exceptions** in Python. They are everywhere.

Virtually every module in the standard Python library uses them, and Python itself will raise them in many different circumstances.

Among the most commonly occurring types of exceptions are `TypeError`, `NameError`, and `ValueError`

```
a = 2 + "hello"
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-16-571bf8afc1b2> in <module>
----> 1 a = 2 + "hello"

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Exception

Something that is not normal

There is nothing rare about **exceptions** in Python. They are everywhere.

Virtually every module in the standard Python library uses them, and Python itself will raise them in many different circumstances.

Among the most commonly occurring types of exceptions are `TypeError`, `NameError`, and `ValueError`

```
print(myvar)

-----
NameError                                 Traceback (most recent call last)
<ipython-input-17-a2ef32bf878c> in <module>
      1 print(myvar)

NameError: name 'myvar' is not defined
```

Exception

Something that is not normal

There is nothing rare about **exceptions** in Python. They are everywhere.

Virtually every module in the standard Python library uses them, and Python itself will raise them in many different circumstances.

Among the most commonly occurring types of exceptions are `TypeError`, `NameError`, and `ValueError`

```
a = float("hello")  
-----  
ValueError                                                 Traceback (most recent call last)  
<ipython-input-18-eb2ffdb98be9> in <module>  
----> 1 a = float("hello")  
  
ValueError: could not convert string to float: 'hello'
```

Handling Exceptions

When an exception is raised that causes the program to terminate,
we say that an **unhandled exception** has been raised.

Handling Exceptions

When an exception is raised that causes the program to terminate,
we say that an **unhandled exception** has been raised.

An exception does not need to lead to program termination.
Exceptions, when raised, can and should be **handled** by the program.

Many times, an exception is something the programmer can and should anticipate.

Handling Exceptions

When an exception is raised that causes the program to terminate, we say that an **unhandled exception** has been raised.

If you know that a line of code might raise an exception when executed, you should handle the exception.

For instance;

if you are trying to read a file, which may not exists there...

You anticipate the possibility of an error while writing your code and include necessary codes there to handle possible exceptions

Handling Exceptions

```
b = int(input())
c = int(input())
a = b/float(c)
print('The success/failure ratio is', a)
print('Now here')
```

```
10
5
The success/failure ratio is 2.0
Now here
```

Handling Exceptions

```
b = int(input())
c = int(input())
a = b/float(c)
print('The success/failure ratio is', a)
print('Now here')
```

10
→ 0

```
-----
ZeroDivisionError                                Traceback (most recent call last)
<ipython-input-5-d92111660837> in <module>
      1 b = int(input())
      2 c = int(input())
----> 3 a = b/float(c)
      4 print('The success/failure ratio is', a)
      5 print('Now here')

ZeroDivisionError: float division by zero
```

Handling Exceptions

```
b = int(input())
c = int(input())
try:
    a = b/float(c)
    print('The success/failure ratio is', a)
except:
    print('Second input can not be zero.ignoring..')
print('Execution goes on as normal...')
```

10

→ 0

Second input can not be zero.ignoring..

Execution goes on as normal...

Handling Exceptions

```
b = int(input())
c = int(input())
try:
    a = b/float(c)
    print('The success/failure ratio is', a)
except:
    print('Second input can not be zero.ignoring..')
print('Execution goes on as normal...')
```

10

5

The success/failure ratio is 2.0

Execution goes on as normal...

Handling Exceptions

```
def sumDigits(s):
    """Assumes s is a string
    Returns the sum of the decimal digits in s
    For example, if s is 'a2b3c' it returns 5"""
    sum = 0
    for i in s:
        sum += int(i)

    return sum

sumDigits("1234")
```

Handling Exceptions

```
def sumDigits(s):
    """Assumes s is a string
    Returns the sum of the decimal digits in s
    For example, if s is 'a2b3c' it returns 5"""
    sum = 0
    for i in s:
        sum += int(i)

    return sum
```

```
sumDigits("1x3y5")
```

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-11-5f5b9ebbd9ac> in <module>
----> 1 sumDigits("1x3y5")

<ipython-input-9-26fc7bf6a33c> in sumDigits(s)
      5     sum = 0
      6     for i in s:
----> 7         sum += int(i)
      8
      9     return sum

ValueError: invalid literal for int() with base 10: 'x'
```

Handling Exceptions

```
def sumDigits(s):
    """Assumes s is a string
    Returns the sum of the decimal digits in s
    For example, if s is 'a2b3c' it returns 5"""
    sum = 0
    for i in s:
        try:
            sum += int(i)
        except ValueError:
            continue

    return sum
```

```
sumDigits("1x3y5")
```

Handling Exceptions

```
def sumDigits(s):
    """Assumes s is a string
    Returns the sum of the decimal digits in s
    For example, if s is 'a2b3c' it returns 5"""
    sum = 0
    for i in s:
        try:
            sum += int(i)
        except ValueError:
            continue

    return sum
```

```
sumDigits("1x3y5")
```

Handling Exceptions

```
def sumDigits(s):
    """Assumes s is a string
    Returns the sum of the decimal digits in s
    For example, if s is 'a2b3c' it returns 5"""
    sum = 0
    for i in s:
        try:
            sum += int(i)
        except:
            continue

    return sum
```

```
sumDigits("1x3y5")
```

Handling Exceptions

```
val = int(input("Enter an integer value:"))
print(val**2)
```

```
Enter an integer value:10
100
```

Handling Exceptions

```
val = int(input("Enter an integer value:"))
print(val**2)
```

```
Enter an integer value:abc
```

```
-----
ValueError                                     Traceback (most recent call last)
<ipython-input-17-848bbde5a9b3> in <module>
----> 1 val = int(input("Enter an integer value:"))
      2 print(val**2)

ValueError: invalid literal for int() with base 10: 'abc'
```

Your program is assuming user will specify what you want all the time!

Handling Exceptions

```
while True:  
    try:  
        val = int(input("Enter an integer value:"))  
        break  
    except ValueError:  
        print("Please provide a valid integer value")  
  
print(val**2)
```

Handling Exceptions

```
while True:  
    try:  
        val = int(input("Enter an integer value:"))  
        break  
    except ValueError:  
        print("Please provide a valid integer value")  
  
print(val**2)
```

```
Enter an integer value:abc  
Please provide a valid integer value  
Enter an integer value:klm  
Please provide a valid integer value  
Enter an integer value:123  
15129
```

No matter what text the user enters,
it will not cause an unhandled exception.

Handling Exceptions

The downside of this change is that the program text has grown from two lines to eight.

If there are many places where the user is asked to enter an integer, this can be problematical. Of course, this problem can be solved by introducing a function:

```
def readInt():
    while True:
        val = input('Enter an integer: ')
        try:
            val = int(val)
            return val
        except ValueError:
            print(val, 'is not an integer')
```

Handling Exceptions

The downside of this change is that the program text has grown from two lines to eight.

If there are many places where the user is asked to enter an integer, this can be problematical. Of course, this problem can be solved by introducing a function:

```
def readInt():
    while True:
        val = input('Enter an integer: ')
        try:
            val = int(val)
            return val
        except ValueError:
            print(val, 'is not an integer')
```

```
a = readInt()
```

```
Enter an integer: aaa
aaa is not an integer
Enter an integer: 124
```

Handling Exceptions

Better yet, this function can be generalized to ask for any type of input

```
def readVal(valtype, requestMsg, errorMsg):
    while True:
        val = input(requestMsg)
        try:
            val = valtype(val)
            return val
        except ValueError:
            print(val, errorMsg)

a = readVal(int,"Enter an int:","not an int")
b = readVal(float,"Enter a float:","not a float")
```

Handling Exceptions

Better yet, this function can be generalized to ask for any type of input

```
def readVal(valtype, requestMsg, errorMsg):
    while True:
        val = input(requestMsg)
        try:
            val = valtype(val)
            return val
        except ValueError:
            print(val, errorMsg)

a = readVal(int,"Enter an int:","not an int")
b = readVal(float,"Enter a float:","not a float")
```

```
Enter an int:5.6
5.6 not an int
Enter an int:356
Enter a float:aaa
aaa not a float
Enter a float:3.768
```

Handling Exceptions

Better yet, this function can be generalized to ask for any type of input

```
def readVal(valtype, requestMsg, errorMsg):
    while True:
        val = input(requestMsg)
        try:
            val = valtype(val)
            return val
        except ValueError:
            print(val, errorMsg)

a = readVal(int,"Enter an int:","not an int")
b = readVal(float,"Enter a float:","not a float")
```

readVal function is polymorphic:
it can work with different arg. types

```
Enter an int:5.6
5.6 not an int
Enter an int:356
Enter a float:aaa
aaa not a float
Enter a float:3.768
```

Note that: int("5.6") -> raises an exception here..

It's not int(5.6)

Handling Exceptions

If it is possible for a block of program code to raise more than one kind of exception, the reserved word `except` can be followed by a tuple of exceptions, e.g.,

```
except (ValueError, TypeError):
```

in which case the `except` block will be entered if any of the listed exceptions is raised within the `try` block.

Handling Exceptions

If it is possible for a block of program code to raise more than one kind of exception, the reserved word `except` can be followed by a tuple of exceptions, e.g.,

```
except (ValueError, TypeError):
```

in which case the `except` block will be entered if any of the listed exceptions is raised within the `try` block.

Alternatively;

we can write a separate `except` block for each kind of exception, which allows the program to choose an action based upon which exception was raised.
If the programmer writes

```
except:
```

the `except` block will be entered if any kind of exception is raised within the `try` block.

Some Built-in Exceptions

```
try:  
    print (a)  
except NameError:  
    print ("NameError: name 'ans' is not defined")  
else:  
    print ("No error!")
```

```
NameError: name 'ans' is not defined
```

Some Built-in Exceptions

```
try:  
    a = 2  
    b = "test"  
    c = a + b  
except TypeError:  
    print ('TypeError Exception Raised')  
else:  
    print ('No error!')
```

TypeError Exception Raised

Some Built-in Exceptions

```
try:  
    a = {1:'a', 2:'b', 3:'c'}  
    print (a[4])  
except LookupError:  
    print ("Key Error Exception Raised.")  
else:  
    print ("No error!")
```

Key Error Exception Raised.

Raising exceptions

In many programming languages, the standard approach to dealing with errors is to have functions return a value (often something analogous to Python's `None`) indicating that something has gone amiss.

Each function invocation has to check whether that value has been returned.

Raising exceptions

In many programming languages, the standard approach to dealing with errors is to have functions return a value (often something analogous to Python's `None`) indicating that something has gone amiss.

Each function invocation has to check whether that value has been returned.

In Python, it is more usual to have [a function raise an exception](#) when it cannot produce a result that is consistent with the function's specification.

Raising exceptions: raise statement

The Python **raise** statement forces a specified exception to occur.

The form of a raise statement is :

raise exceptionName(arguments)

The *exceptionName* is usually one of the built-in exceptions, e.g., NameError.

```
try:  
    raise NameError('HiThere')  
except NameError:  
    print('An exception flew by!')  
    raise
```

Raising exceptions: raise statement

```
try:  
    raise NameError('HiThere')  
except NameError:  
    print('An exception flew by!')  
    raise
```

An exception flew by!

```
NameError                                  Traceback (most recent call last)  
<ipython-input-11-bf6ef4926f8c> in <module>  
      1 try:  
----> 2     raise NameError('HiThere')  
      3 except NameError:  
      4     print('An exception flew by!')  
      5     raise  
  
NameError: HiThere
```

Raising exceptions: raise statement

```
try:  
    raise NameError('HiThere')  
except NameError:  
    print('An exception flew by!')  
raise
```

An exception flew by!

```
RuntimeError                                Traceback (most recent call last)
```

```
<ipython-input-12-41d607222dd2> in <module>  
      3 except NameError:  
      4     print('An exception flew by!')  
----> 5 raise
```

```
RuntimeError: No active exception to reraise
```

Raising exceptions: raise statement

```
def f():
    → raise Exception()

def g():
    try:
        f()
    except:
        print("caught an exception in f()")
    → raise

def h():
    try:
        g()
    except:
        print("caught an exception in g()")

h()
```

Raising exceptions: raise statement

```
def f():
    raise Exception()

def g():
    try:
        f()
    except:
        print("caught an exception in f()")
        raise
def h():
    try:
        g()
    except:
        print("caught an exception in g()")

h()
```

```
caught an exception in f()
caught an exception in g()
```

Raising exceptions: raise statement

```
def f():
    raise Exception("Dummy exception!")

def g():
    try:
        f()
    except:
        print("caught an exception in f()")
        raise

def h():
    try:
        g()
    except:
        print("caught an exception in g()")
        → raise

h()
```

Raising exceptions: raise statement

```
def f():
    raise Exception("Dummy exception!")

def g():
    try:
        f()
    except:
        print("caught an exception in f()")
        raise

def h():
    try:
        g()
    except:
        print("caught an exception in g()")
    → raise

h()
```

```
caught an exception in f()
caught an exception in g()

-----
Exception                                         Traceback (most recent call last)
<ipython-input-23-61a54d88ff85> in <module>
      15         raise
      16
---> 17 h()

<ipython-input-23-61a54d88ff85> in h()
      10 def h():
      11     try:
---> 12         g()
      13     except:
      14         print("caught an exception in g()")

<ipython-input-23-61a54d88ff85> in g()
      4 def g():
      5     try:
---> 6         f()
      7     except:
      8         print("caught an exception in f()")

<ipython-input-23-61a54d88ff85> in f()
      1 def f():
---> 2     raise Exception("Dummy exception!")
      3
      4 def g():
      5     try:

Exception: Dummy exception!
```

Assertions

The Python `assert` statement provides programmers with a simple way to confirm that the state of the computation is as expected.

An `assert` statement can take one of two forms:

assert Boolean expression

or

assert Boolean expression, argument

When an `assert` statement is encountered, the Boolean expression is evaluated;

- > If it evaluates to `True`, execution proceeds in a normal way.
- > If it evaluates to `False`, an `AssertionError` exception is raised.

Assertions

```
def KelvinToFahrenheit(Temperature):
    assert (Temperature >= 0)
    return ((Temperature-273)*1.8)+32

print(KelvinToFahrenheit(273))
print(KelvinToFahrenheit(-5))
```

Assertions

```
def KelvinToFahrenheit(Temperature):
    assert (Temperature >= 0)
    return ((Temperature-273)*1.8)+32
```

```
print(KelvinToFahrenheit(273))
print(KelvinToFahrenheit(-5))
```

```
32.0
```

```
-----
AssertionError                                     Traceback (most recent call last)
<ipython-input-15-b84d744169ab> in <module>
      4
      5     print(KelvinToFahrenheit(273))
----> 6     print(KelvinToFahrenheit(-5))

<ipython-input-15-b84d744169ab> in KelvinToFahrenheit(Temperature)
      1 def KelvinToFahrenheit(Temperature):
----> 2     assert (Temperature >= 0)
      3     return ((Temperature-273)*1.8)+32
      4
      5 print(KelvinToFahrenheit(273))

AssertionError:
```

Assertions

```
def KelvinToFahrenheit(Temperature):
    assert (Temperature >= 0), "Colder than absolute zero!"
    return ((Temperature-273)*1.8)+32

print(KelvinToFahrenheit(273))
print(KelvinToFahrenheit(-5))
```

Assertions

```
def KelvinToFahrenheit(Temperature):
    assert (Temperature >= 0), "Colder than absolute zero!"
    return ((Temperature-273)*1.8)+32

print(KelvinToFahrenheit(273))
print(KelvinToFahrenheit(-5)) 32.0

-----
AssertionError                                         Traceback (most recent call last)
<ipython-input-14-f1457cda2a14> in <module>
      4
      5     print(KelvinToFahrenheit(273))
----> 6     print(KelvinToFahrenheit(-5))

<ipython-input-14-f1457cda2a14> in KelvinToFahrenheit(Temperature)
      1 def KelvinToFahrenheit(Temperature):
----> 2     assert (Temperature >= 0),"Colder than absolute zero!"
      3     return ((Temperature-273)*1.8)+32
      4
      5     print(KelvinToFahrenheit(273))

AssertionError: Colder than absolute zero!
```

Lecture 13

Modules and Files

Dr. Hacer Yalım Keleş

hkeles@ankara.edu.tr

Disclaimer: Content of this lecture slides are mostly from **Section 4.5 and 4.6** of Guttag's Book: "*Introduction to Computation Programming Using Python*".

Modules

```
x Circle.py x
pi = 3.14159

def area(radius):
    return pi*(radius**2)

def circumference(radius):
    return 2*pi*radius

def sphereSurface(radius):
    return 4.0*area(radius)

def sphereVolume(radius):
    return (4.0/3.0)*pi*(radius**3)
```

Circle.py is a module

Modules

```
x Circle.py x
pi = 3.14159

def area(radius):
    return pi*(radius**2)

def circumference(radius):
    return 2*pi*radius

def sphereSurface(radius):
    return 4.0*area(radius)

def sphereVolume(radius):
    return (4.0/3.0)*pi*(radius**3)
```

```
main.py x
import Circle

radius = 2.3
print("%.2f" %(Circle.pi))
a = Circle.area(radius)
b = Circle.circumference(radius)
print("%.2f\t%.2f" %(a,b))
```

Circle.py is a module

Modules

- Modules are typically stored in individual files.
- Each module has its own private symbol table.
 - Within `circle.py` we access objects (e.g., `pi` and `area`) in the usual way.
- Executing “`import M`” creates a binding for module `M` in the scope in which the importation occurs.
 - Therefore, in the importing context we use dot notation to indicate that we are referring to a name defined in the imported module.
- For example, outside of `circle.py`, the references `pi` and `circle.pi` can refer to different objects.

Modules

There is a variant of the import statement that allows the importing program to omit the module name when accessing names defined inside the imported module.

Modules

```
x Circle.py x
pi = 3.14159

def area(radius):
    return pi*(radius**2)

def circumference(radius):
    return 2*pi*radius

def sphereSurface(radius):
    return 4.0*area(radius)

def sphereVolume(radius):
    return (4.0/3.0)*pi*(radius**3)
```

```
main.py x
import Circle

radius = 2.3
print("%.2f" %(Circle.pi))
a = Circle.area(radius)
b = Circle.circumference(radius)
print("%.2f\t%.2f" %(a,b))
```

Output

```
3.14
16.62 14.45
```

Circle.py is a module

Modules

```
x Circle.py x
pi = 3.14159

def area(radius):
    return pi*(radius**2)

def circumference(radius):
    return 2*pi*radius

def sphereSurface(radius):
    return 4.0*area(radius)

def sphereVolume(radius):
    return (4.0/3.0)*pi*(radius**3)
```

```
main.py x
from Circle import *

radius = 2.3
print("%.2f" %(pi))
a = area(radius)
b = circumference(radius)
print("%.2f\t%.2f" %(a,b))
```

Output

```
3.14
16.62 14.45
```

Circle.py is a module

Modules

- A module can contain executable statements as well as function definitions. Typically, these statements are used to initialize the module.
- The statements in a module are executed only the first time a module is imported into a program.

Files

- Each operating system (e.g., Windows and MAC OS) comes with its own file system for creating and accessing files.
- Python achieves operating-system independence by accessing files through something called a **file handle**.
- The code:

```
fHandle = open('myfile', 'w')
```

—>instructs the operating system to create a file with the name my file, and return a file handle for that file.

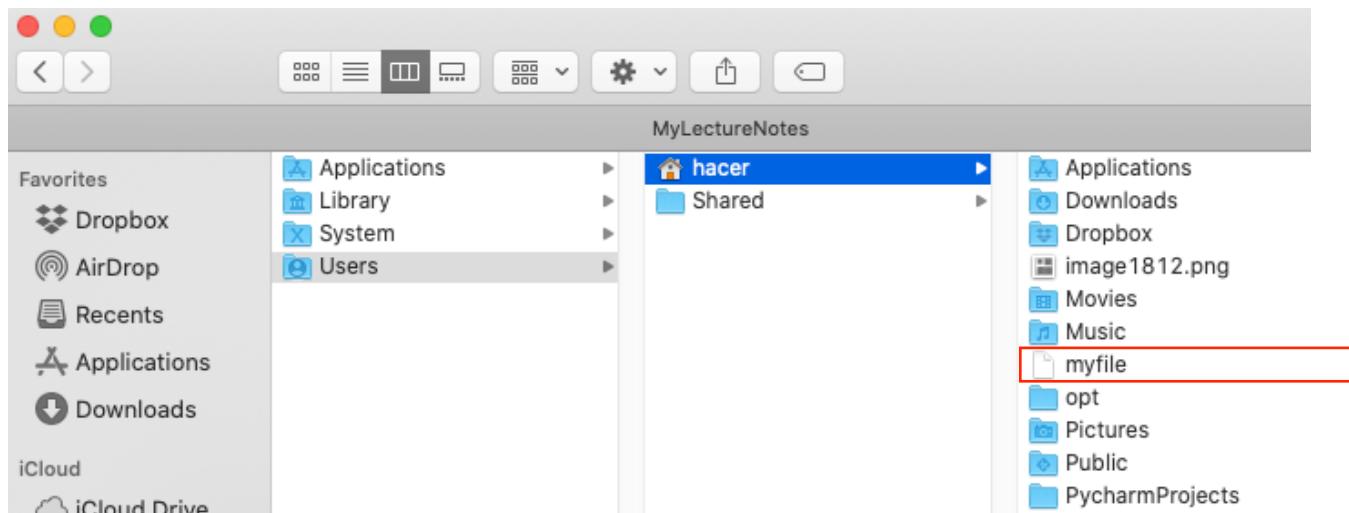
- You can use the **write** method to write into the files opened
- It is important to remember to close the file when the program is finished using it. Otherwise there is a risk that some or all of the writes may not be saved.

Writing into Files

```
fHandle = open('/Users/hacer/myfile', 'w')
for i in range(4):
    name = input('Enter name: ')
    fHandle.write(name + '\n')
fHandle.close()
```

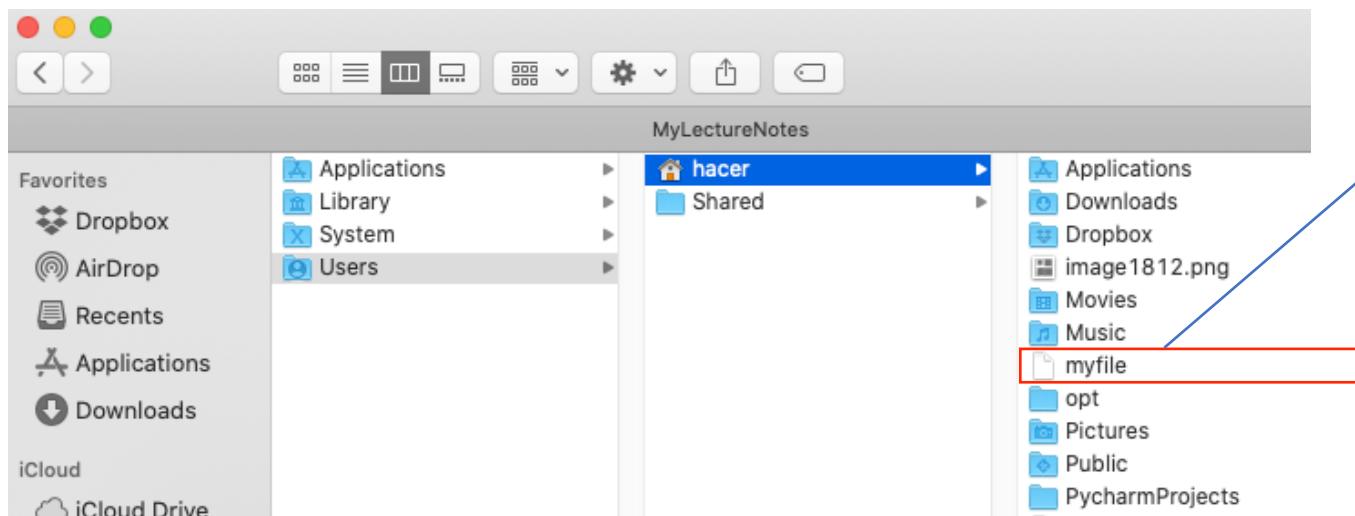
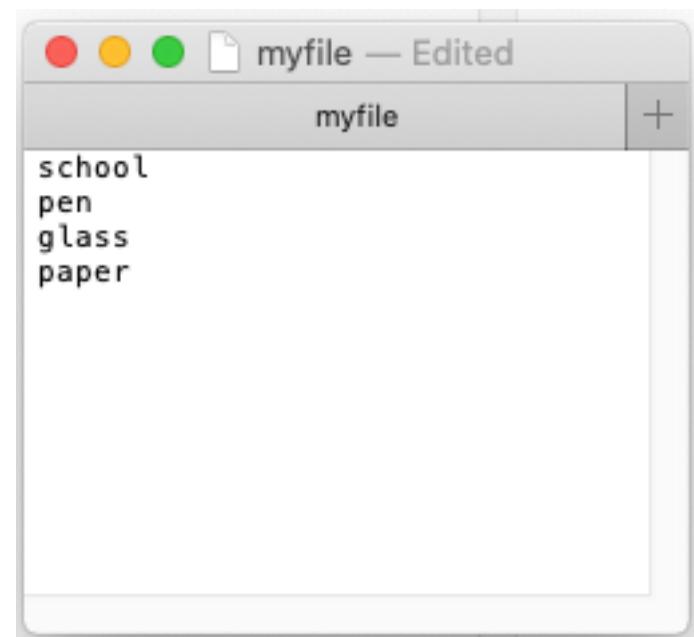
Writing into Files

```
fHandle = open('/Users/hacer/myfile', 'w')
for i in range(4):
    name = input('Enter name: ')
    fHandle.write(name + '\n')
fHandle.close()
```



Writing into Files

```
fHandle = open('/Users/hacer/myfile', 'w')
for i in range(4):
    name = input('Enter name: ')
    fHandle.write(name + '\n')
fHandle.close()
```



Reading from Files

```
fHandle = open(filename, 'r')  
for line in fHandle:  
    print(line)  
  
fHandle.close()
```

Output:

school

pen

glass

paper



Reading from Files

```
fHandle = open(filename, 'r')  
for line in fHandle:  
    print(line)  
  
fHandle.close()
```

Output:

school



pen

glass

paper

Reading from Files

```
fHandle = open(filename, 'r')
for line in fHandle:
    print(line[:-1])

fHandle.close()
```

Output:

```
school
pen
glass
paper
```

Reading from Files

```
fHandle = open(filename, 'r')
for line in fHandle:
    print(line, end='')

fHandle.close()
```

Output:

```
school
pen
glass
paper
```

Reading from Files

Arranging these calls in functions

```
def writeToFile(filename):
    fHandle = open(filename, 'w')
    for i in range(4):
        name = input('Enter name: ')
        fHandle.write(name + '\n')
    fHandle.close()

def readFromFile(filename):
    fHandle = open(filename, 'r')
    for line in fHandle:
        print(line[:-1])

    fHandle.close()

fname = '/Users/hacer/myfile'
writeToFile(fname)
readFromFile(fname)
```

Appending to Files

```
def appendToFile(filename):
    fHandle = open(filename, 'a')
    for i in range(3):
        name = input('Enter name: ')
        fHandle.write(name + '\n')
    fHandle.close()

fname = '/Users/hacer/myfile'
appendToFile(fname)
readFromFile(fname)
```

Appending to Files

```
def appendToFile(filename):
    fHandle = open(filename, 'a')
    for i in range(3):
        name = input('Enter name: ')
        fHandle.write(name + '\n')
    fHandle.close()

fname = '/Users/hacer/myfile'
appendToFile(fname)
readFromFile(fname)
```

Output:

```
school
pen
glass
paper
bird
lion
clock
```

Appending to Files

```
def readLines(filename):
    fh = open(filename, 'r')
    cont = fh.readlines()
    print(cont)
    fh.close()

fname = '/Users/hacer/myfile'
readLines(fname)
```

Output:

```
['school\n', 'pen\n', 'glass\n', 'paper\n', 'bird\n', 'lion\n', 'clock\n']
```

Appending to Files

```
def readLines(filename):
    fh = open(filename, 'r')
    cont = fh.readlines()
    print(cont)
    fh.close()
    return cont

def writeLines(filename, L):
    fh = open(filename, 'a')
    fh.writelines(L)
    fh.close()

fname = '/Users/hacer/myfile'
L = readLines(fname)
writeLines(fname, L)
```

Appending to Files

```
def readLines(filename):
    fh = open(filename, 'r')
    cont = fh.readlines()
    print(cont)
    fh.close()
    return cont

def writeLines(filename, L):
    fh = open(filename, 'a')
    fh.writelines(L)
    fh.close()

fname = '/Users/hacer/myfile'
L = readLines(fname)
writeLines(fname, L)
```

myfile

school
pen
glass
paper
bird
lion
clock

Commonly Used File Functions

open(fn, 'w') fn is a string representing a file name. Creates a file for writing and returns a file handle.

open(fn, 'r') fn is a string representing a file name. Opens an existing file for reading and returns a file handle.

open(fn, 'a') fn is a string representing a file name. Opens an existing file for appending and returns a file handle.

fh.read() returns a string containing the contents of the file associated with the file handle fh.

fh.readline() returns the next line in the file associated with the file handle fh.

fh.readlines() returns a list each element of which is one line of the file associated with the file handle fh.

fh.write(s) write the string s to the end of the file associated with the file handle fh.

fh.writeLines(S) S is a sequence of strings. Writes each element of S to the file associated with the file handle fh.

fh.close() closes the file associated with the file handle fh.

Lecture 13

ADT and Classes

Dr. Hacer Yalım Keleş

hkeles@ankara.edu.tr

Disclaimer: *Some slide contents are taken from <https://python-textbook.readthedocs.io/en/1.0>*

Abstract Data Types (ADT)

An **abstract data type** is a set of objects and the operations on those objects.

Object-Oriented Programming (OOP)

Everything is object centered.

- The problem and the solution are modeled based on objects.
- The data and the algorithm are combined in objects.

What is an object?

An object is an entity which has a state and a set of behaviors that, when executed, change the state of the entity or the environment.



A car object has:

State:

Position, Speed, Gear State, Brake State, Wheel
State

Behaviors:

Rotate, Accelerate, Brake

Abstract Data Types (ADT)

An **abstract data type** is a set of objects and the operations on those objects.

In Python, one implements data abstractions using **classes**

Defining and Using a Class

```
class <class_name>:  
    <class_data_&_method_definitions>
```

Defining and Using a Class

```
class <class_name>:  
    <class_data_&_method_definitions>
```

One special class method `__init__`:
used to initialize object data

```
def __init__(self, <params>):  
    <initializations>
```

When we call the class object, a new instance of the class is created, and the `__init__` method on this new object is immediately executed with all the parameters that we passed to the class object.

Example Class: Person

```
class Person:

    def __init__(self, name, surname, address, telephone, email):
        self.name = name
        self.surname = surname
        self.address = address
        self.telephone = telephone
        self.email = email

    def print_info(self):
        print(self.name, self.surname)
        print(self.address)
        print("contact:", self.email)
```

Example Class: Person

```
class Person:

    def __init__(self, name, surname, address, telephone, email):
        self.name = name
        self.surname = surname
        self.address = address
        self.telephone = telephone
        self.email = email

    def print_info(self):
        print(self.name, self.surname)
        print(self.address)
        print("contact:", self.email)
```

```
#create a person object
John = Person(
    "John",
    "Simith",
    "Ata Street, Ankara",
    "555 444 00 01",
    "john.simith@blabla.com"
)
#call object method
John.print_info()
```

Example Class: Person

```
class Person:

    def __init__(self, name, surname, address, telephone, email):
        self.name = name
        self.surname = surname
        self.address = address
        self.telephone = telephone
        self.email = email

    def print_info(self):
        print(self.name, self.surname)
        print(self.address)
        print("contact:", self.email)
```

```
#create a person object
John = Person(
    "John",
    "Simith",
    "Ata Street, Ankara",
    "555 444 00 01",
    "john.simith@blabla.com"
)
#call object method
John.print_info()
```

```
John Simith
Ata Street, Ankara
contact: john.simith@blabla.com
```

Example Class: Person

```
class Person:

    def __init__(self, name, surname, address, telephone, email):
        self.name = name
        self.surname = surname
        self.address = address
        self.telephone = telephone
        self.email = email

    def print_info(self):
        print(self.name, self.surname)
        print(self.address)
        print("contact:", self.email)
```

```
#create a person object
John = Person(
    "John",
    "Simith",
    "Ata Street, Ankara",
    "555 444 00 01",
    "john.simith@blabla.com"
)
#call object method
John.print_info()
```

No `self` argument during method call

both of these method definitions have `self` as the first parameter, and we use this variable inside the method bodies; but we don't pass this parameter in.

Whenever we call a method on an object, *the object itself* is automatically passed in as the first parameter. This gives us a way to access the object's properties from inside the object's methods.

Example Class: Person

```
class Person:

    def __init__(self, name, surname, address, telephone, email):
        self.name = name
        self.surname = surname
        self.address = address
        self.telephone = telephone
        self.email = email

    def print_info(self):
        print(self.name, self.surname)
        print(self.address)
        print("contact:", self.email)
```

```
#create a person object
John = Person(
    "John",
    "Simith",
    "Ata Street, Ankara",
    "555 444 00 01",
    "john.simith@blabla.com"
)
#call object method
John.print_info()
```

The `__init__` function creates **attributes** on the object and sets them to the **values** we have passed in as parameters.

Example Class: Person

```
class Person:

    def __init__(self, name, surname, address, telephone, email):
        self.name = name
        self.surname = surname
        self.address = address
        self.telephone = telephone
        self.email = email

    def print_info(self):
        print(self.name, self.surname)
        print(self.address)
        print("contact:", self.email)
```

```
#create a person object
John = Person(
    "John",
    "Simith",
    "Ata Street, Ankara",
    "555 444 00 01",
    "john.simith@blabla.com"
)
#call object method
John.print_info()
```

```
print(John.name)
print(John.surname)
```

John
Simith

Example Class: Person

```
class Person:

    def __init__(self, name, surname, address, telephone, email):
        self.name = name
        self.surname = surname
        self.address = address
        self.telephone = telephone
        self.email = email

    def print_info(self):
        print(self.name, self.surname)
        print(self.address)
        print("contact:", self.email)
```

```
<__main__.Person object at 0x7feac9e90f40>
<class '__main__.Person'>
<class 'method'>
```

```
#create a person object
John = Person(
    "John",
    "Simith",
    "Ata Street, Ankara",
    "555 444 00 01",
    "john.simith@blabla.com"
)
#call object method
John.print_info()
```

```
print(John.name)
print(John.surname)
```

John
Simith

```
print(John)
print(type(John))
print(type(John.print_info))
```

Example Class: Person

```
class Person:  
    ...
```

Definition of Person class

```
def __init__(self, ... ):  
    ...
```

Object initializer.
self: current object - instance

```
def print_info(self):  
    ...
```

Object method - member function
self: current object - instance

```
John = Person(...)
```

An instance of Person class

```
John.name = "Johnny"
```

A member variable of Person object named John, set to "Johnny"
State of the object John has been modified.

Example Class: Person

- Remember that defining a function doesn't make the function run.
- Defining a class also doesn't make anything run – it informs Python about the class.
- The class will not be defined until Python has executed the entirety of the definition
- You can reference any method from any other method on the same class,
- You can even reference the class inside a method of the class.
- By the time you call that method, the entire class will definitely be defined.

Example Class: Point

Point.py

```
from math import sqrt

class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def norm(self):
        return sqrt(self.x*self.x+self.y*self.y)

    def dist(self, p):
        return sqrt((self.x-p.x)**2+(self.y-p.y)**2)
```

Class name

Object initializer

Member functions / methods

Example Class: Point

Point.py

```
from math import sqrt

class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def norm(self):
        return sqrt(self.x*self.x+self.y*self.y)

    def dist(self, p):
        return sqrt((self.x-p.x)**2+(self.y-p.y)**2)
```

Main.py

```
from Point import *

p1 = Point(1,0)
p2 = Point(3,0)
print(p1.norm())
print(p1.dist(p2))
```

Output 1.0
2.0

Lecture 16

Objects & Graphics

Dr. Hacer Yalım Keleş

hkeles@ankara.edu.tr

Disclaimer:The content is based on Chapter 4 of ***John Zelle's Book: "Python Programming: An Introduction to Computer Science" (Third Edn.)***

graphics.py

A simple graphics module that we'll be experimenting with in this lecture.
We'll see the benefits of OOP while learning a graphics package..

graphics.py

A simple graphics module that we'll be experimenting with in this lecture.
We'll see the benefits of OOP while learning a graphics package..

You can;

- copy `graphics.py` to the directory where you run your programs
- place it in a system directory where other Python libraries are stored

Using graphics module

From python shell:

```
>>>import graphics  
>>>win = graphics.GraphWin()  
>>>win.close()
```

Using graphics module

From python shell:

```
>>>import graphics  
>>>win = graphics.GraphWin("My window")  
>>>win.close()
```



A graphics window is a collection of tiny points, called pixels
By changing the pixel colors we can control the content of the window
By default, GraphWin creates a window of 200x200 pixels
Changing individual pixels (40K) is challenging; so we will use a library of graphical objects.
Each object has related functions to draw themselves into a GraphWin...

Point objects

In geometry, a point is a location in space.

A reference coordinate system is required for that

GraphWin defines a cartesian coordinate system, where the origin (0,0) is assumed at the top left corner of the window.

x increases from left-to-right

y increases from top-to-bottom

We'll define Points using (x, y) coordinates accordingly;

x: horizontal location, y: vertical location

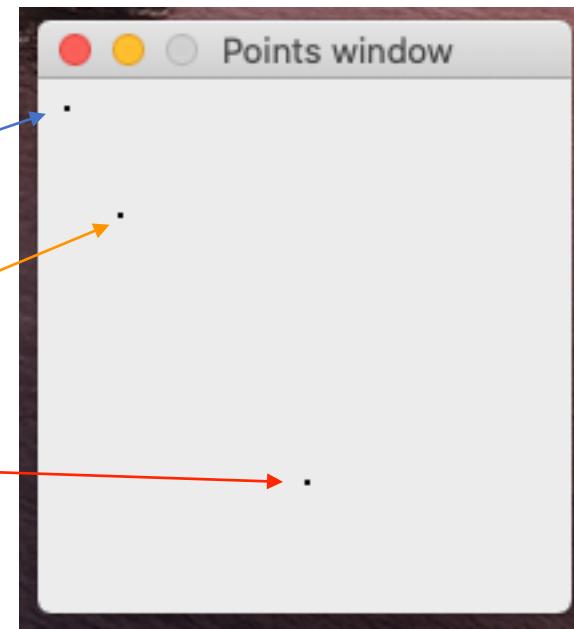
Drawing a point changes the pixel at that point; the default color is black.

Point objects

```
>>> from graphics import *
>>> p1 = Point(10,10)
>>> p2 = Point(100,150)
>>> p3 = Point(30,50)
>>> win = GraphWin("Points window")
>>> p1.draw(win)
Point(10.0, 10.0)
>>> p2.draw(win)
Point(100.0, 150.0)
>>> p3.draw(win)
Point(30.0, 50.0)
```

Point objects

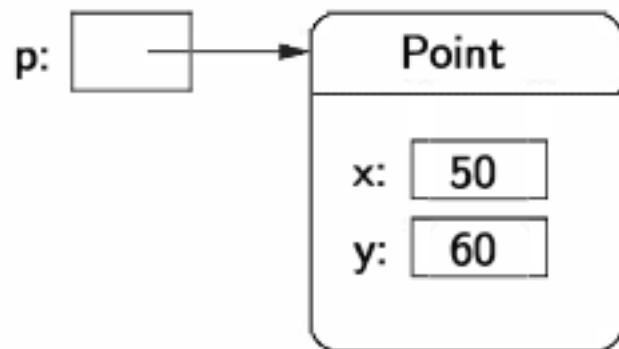
```
>>> from graphics import *
>>> p1 = Point(10,10)
>>> p2 = Point(100,150)
>>> p3 = Point(30,50)
>>> win = GraphWin("Points window")
>>> p1.draw(win)
Point(10.0, 10.0)
>>> p2.draw(win)
Point(100.0, 150.0)
>>> p3.draw(win)
Point(30.0, 50.0)
```



Point objects

```
p = Point(50,60)
```

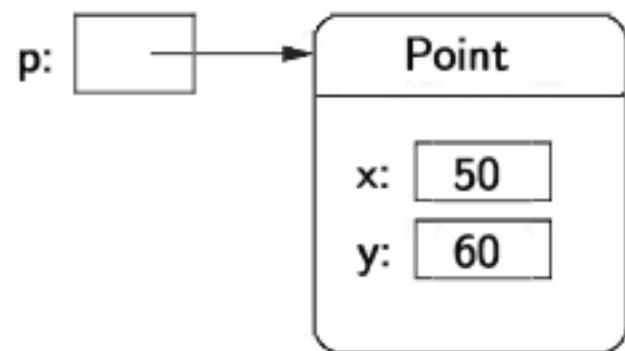
The constructor for the `Point` class requires two parameters giving the *x* and *y* coordinates for the new point. These values are stored as *instance variables* inside the object. In this case, Python creates an instance of `Point` having an *x* value of 50 and a *y* value of 60. The resulting point is then assigned to the variable `p`.



Point objects

```
p.getX()  
p.getY()
```

The `getX` and `getY` methods return the x and y values of a point, respectively. Methods such as these are sometimes called *accessors*, because they allow us to access information from the instance variables of the object.



Point objects

```
p.move(10,0)
```

This changes the `x` instance variable of `p` by adding 10 units. If the point is currently drawn in a `GraphWin`, `move` will also take care of erasing the old image and drawing it in its new position. Methods that change the state of an object are sometimes called *mutators*.

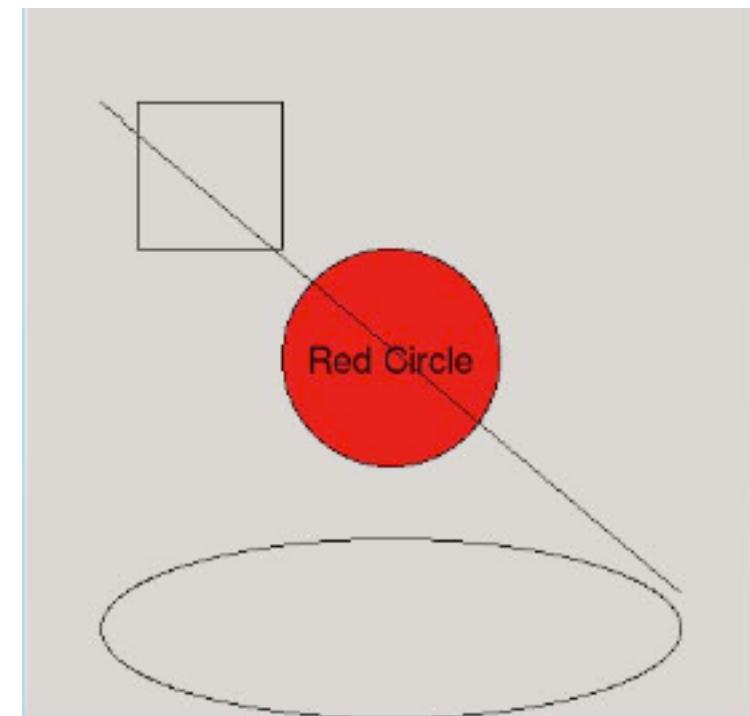
Other objects

There are other objects like lines, circles, rectangles, ovals, polygons and text

Other objects

There are other objects like lines, circles, rectangles, ovals, polygons and text

```
>>> ##### Open a graphics window
>>> win = GraphWin('Shapes')
>>> ##### Draw a red circle centered at point (100,100) with radius 30
>>> center = Point(100,100)
>>> circ = Circle(center, 30)
>>> circ.setFill('red')
>>> circ.draw(win)
>>> ##### Put a textual label in the center of the circle
>>> label = Text(center, "Red Circle")
>>> label.draw(win)
>>> ##### Draw a square using a Rectangle object
>>> rect = Rectangle(Point(30,30), Point(70,70))
>>> rect.draw(win)
>>> ##### Draw a line segment using a Line object
>>> line = Line(Point(20,30), Point(180, 165))
>>> line.draw(win)
>>> ##### Draw an oval using the Oval object
>>> oval = Oval(Point(20,150), Point(180,199))
>>> oval.draw(win)
```



Other objects

```
circ = Circle(Point(100,100), 30)
win = GraphWin()
circ.draw(win)
```

Center position

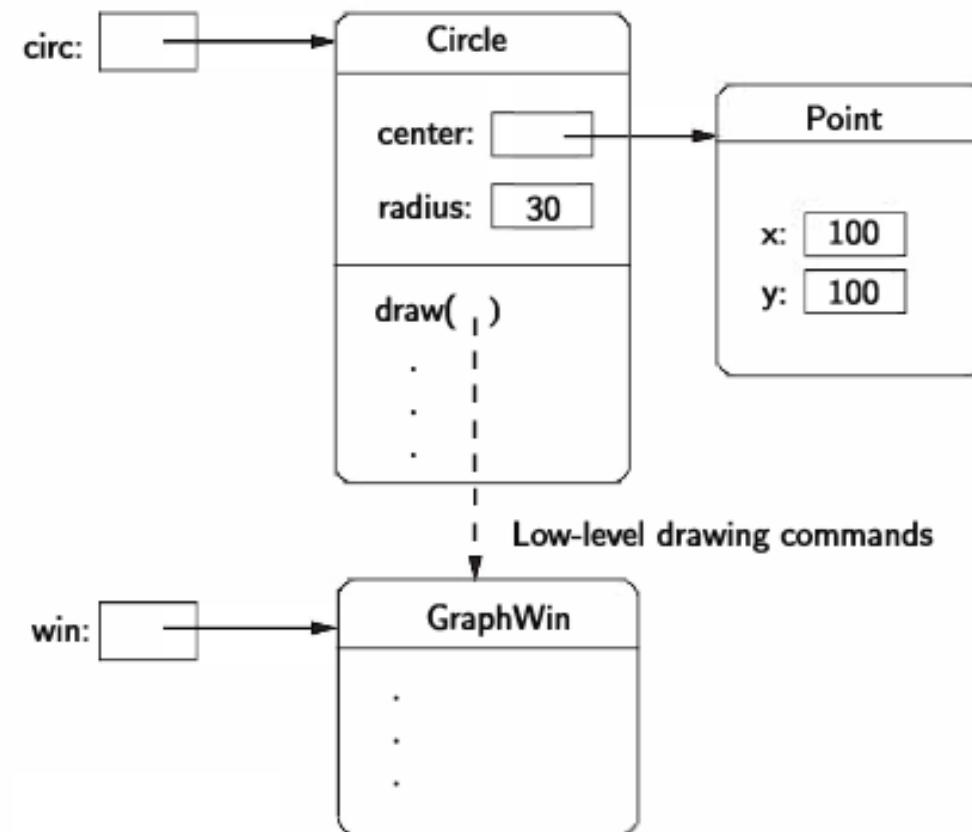
Radius

Other objects

```
circ = Circle(Point(100,100), 30)
win = GraphWin()
circ.draw(win)
```

Center position

Radius



Object interactions to draw a circle

Aliasing (recap)

```
class mydummy:  
    def __init__(self, a):  
        self.a = a  
  
    def print(self):  
        print(self.a)  
  
    def seta(self, a):  
        self.a = a  
  
d1 = mydummy(3)  
d2 = d1  
d2.seta(5)  
d1.print()  
print(d1 is d2)
```

Aliasing

```
class mydummy:  
    def __init__(self, a):  
        self.a = a  
  
    def print(self):  
        print(self.a)  
  
    def seta(self, a):  
        self.a = a  
  
d1 = mydummy(3)  
d2 = d1  
d2.seta(5)  
d1.print()  
print(d1 is d2)
```

User defined types, classes, are **mutable**

To simulate immutability in a class, one should override attribute setting and deletion to raise exceptions.

Aliasing may sometimes create unexpected results...
Let's see an example.

5
True

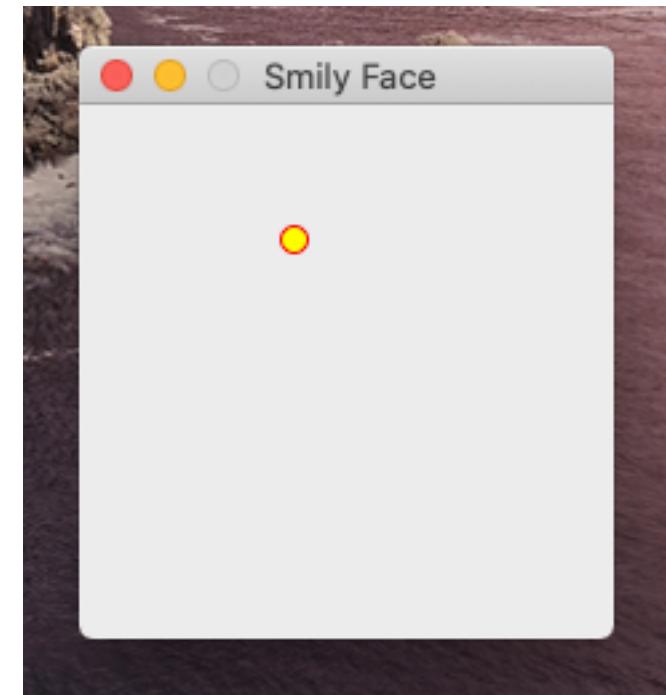
Aliasing

```
from graphics import *

def drawSmilyface():
    win = GraphWin("Smily Face")
    leftEye = Circle(Point(80, 50), 5)
    leftEye.setFill('yellow')
    leftEye.setOutline('red')
    leftEye.draw(win)

    win.getMouse()

if __name__ == '__main__':
    drawSmilyface()
```



Aliasing

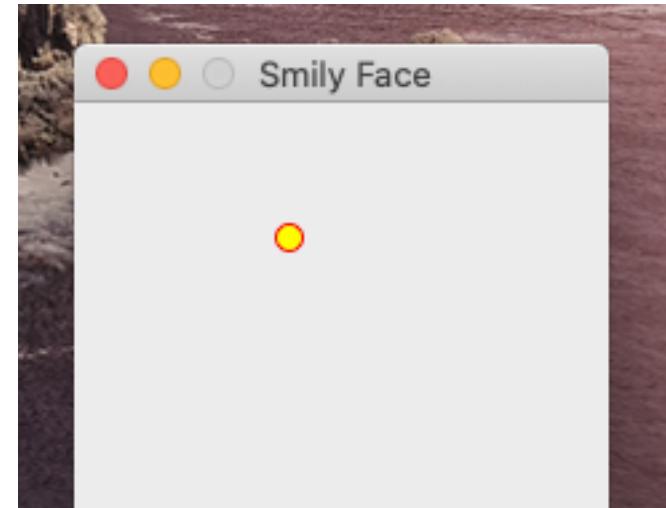
MyGUI.py

```
from graphics import *

def drawSmilyface():
    win = GraphWin("Smily Face")
    leftEye = Circle(Point(80, 50), 5)
    leftEye.setFill('yellow')
    leftEye.setOutline('red')
    leftEye.draw(win)

    win.getMouse()

if __name__ == '__main__':
    drawSmilyface()
```



Every python module has a global `__name__` variable, set by Python interpreter
if you run MyGUI.py module as your primary module like:
`>>>python MyGUI.py`
Global variable `__name__` is set to '`__main__`'
if you import MyGUI in another module,
`__name__` is set to 'MyGUI' —> module name

Aliasing

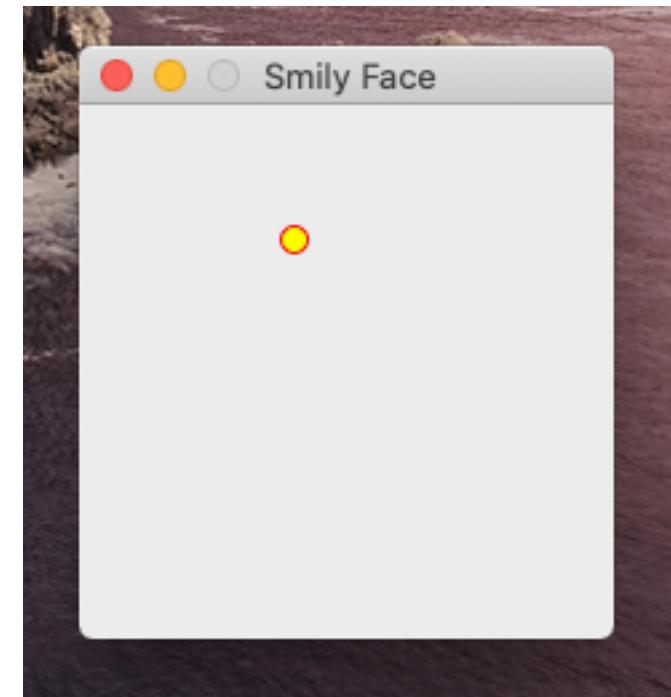
MyGUI.py

```
from graphics import *

def drawSmilyface():
    win = GraphWin("Smily Face")
    leftEye = Circle(Point(80, 50), 5)
    leftEye.setFill('yellow')
    leftEye.setOutline('red')
    leftEye.draw(win)

    win.getMouse()

if __name__ == '__main__':
    drawSmilyface()
```



GraphWin will close after getting a mouse click

Aliasing

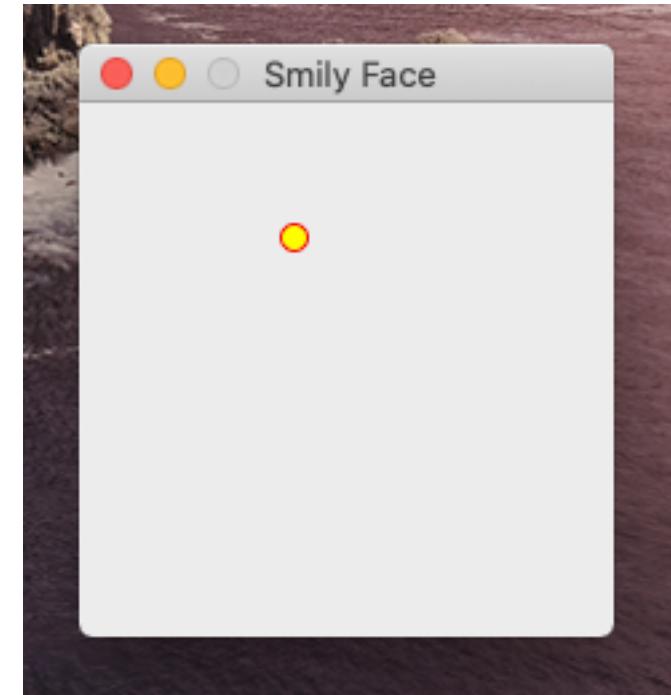
MyGUI.py

```
from graphics import *

def drawSmilyface():
    win = GraphWin("Smily Face")
    leftEye = Circle(Point(80, 50), 5)
    leftEye.setFill('yellow')
    leftEye.setOutline('red')
    leftEye.draw(win)

    win.getMouse()

if __name__ == '__main__':
    drawSmilyface()
```



Let us try to draw the right eye simply copying the left eye and moving it a little bit to the right..quite simple right?

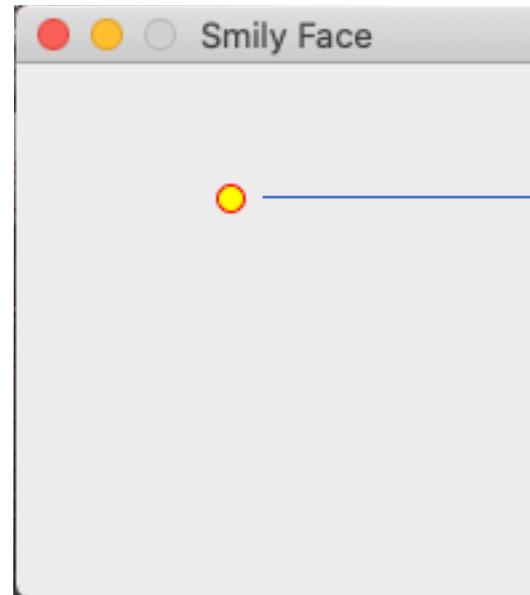
Aliasing

MyGUI.py

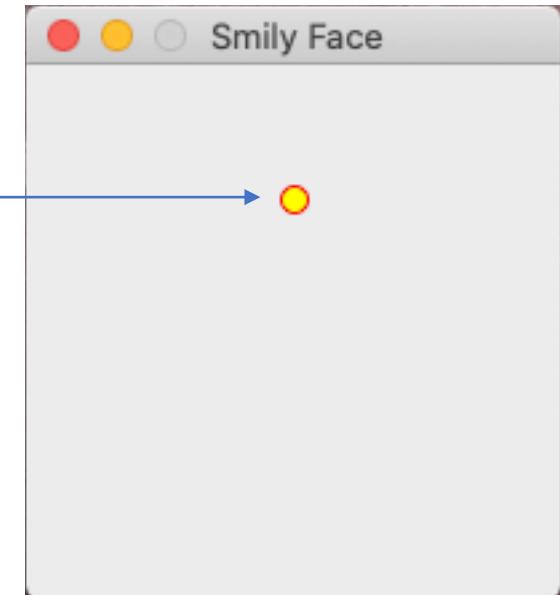
```
def drawSmilyface():
    win = GraphWin("Smily Face")
    leftEye = Circle(Point(80, 50), 5)
    leftEye.setFill('yellow')
    leftEye.setOutline('red')
    leftEye.draw(win)
    rightEye = leftEye
    rightEye.move(20,0)

    win.getMouse()
```

Left eye



Right eye



What happened to left eye?

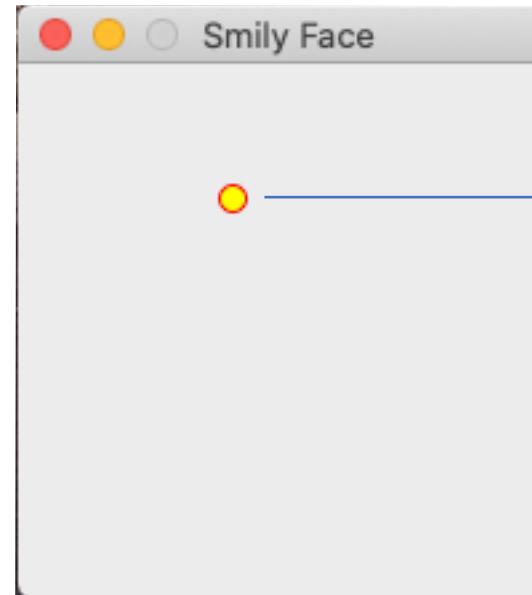
Aliasing

MyGUI.py

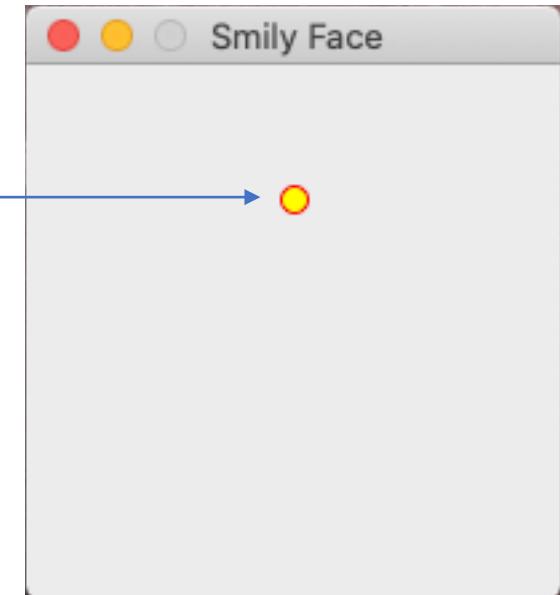
```
def drawSmilyface():
    win = GraphWin("Smily Face")
    leftEye = Circle(Point(80, 50), 5)
    leftEye.setFill('yellow')
    leftEye.setOutline('red')
    leftEye.draw(win)
    rightEye = leftEye
    rightEye.move(20,0)

    win.getMouse()
```

Left eye



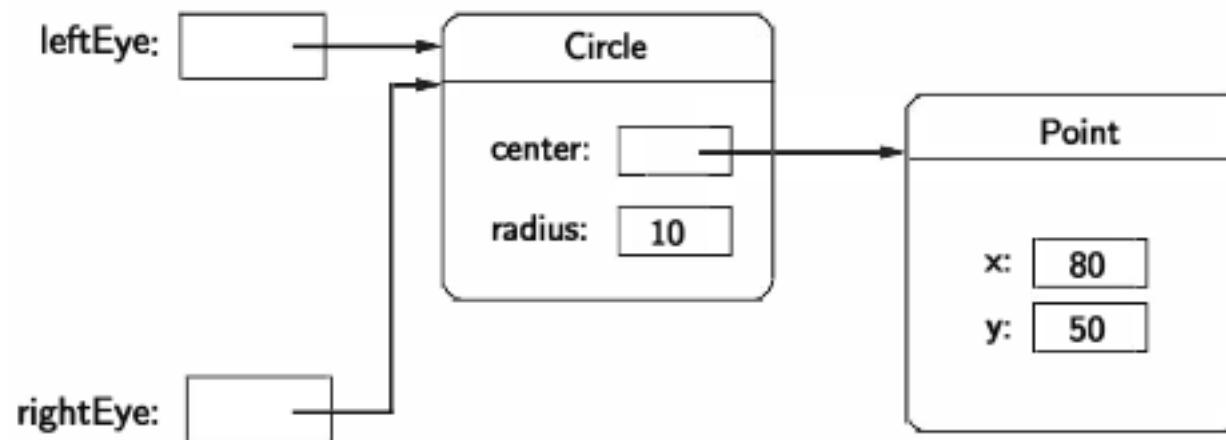
Right eye



What happened to left eye?

There is just one object indeed...Assignment created aliasing

Aliasing



Aliasing

```
win = GraphWin("Smily Face")
leftEye = Circle(Point(80, 50), 5)
leftEye.setFill('yellow')
leftEye.setOutline('red')
leftEye.draw(win)
rightEye = Circle(Point(100, 50), 5)
rightEye.setFill('yellow')
rightEye.setOutline('red')
rightEye.draw(win)
```



Create new Circle object

Aliasing

```
win = GraphWin("Smily Face")
leftEye = Circle(Point(80, 50), 5)
leftEye.setFill('yellow')
leftEye.setOutline('red')
leftEye.draw(win)
rightEye = Circle(Point(100, 50), 5)
rightEye.setFill('yellow')
rightEye.setOutline('red')
rightEye.draw(win)
```



Create new Circle object
it is tedious to create very similar object - repeating
similar routines

Aliasing

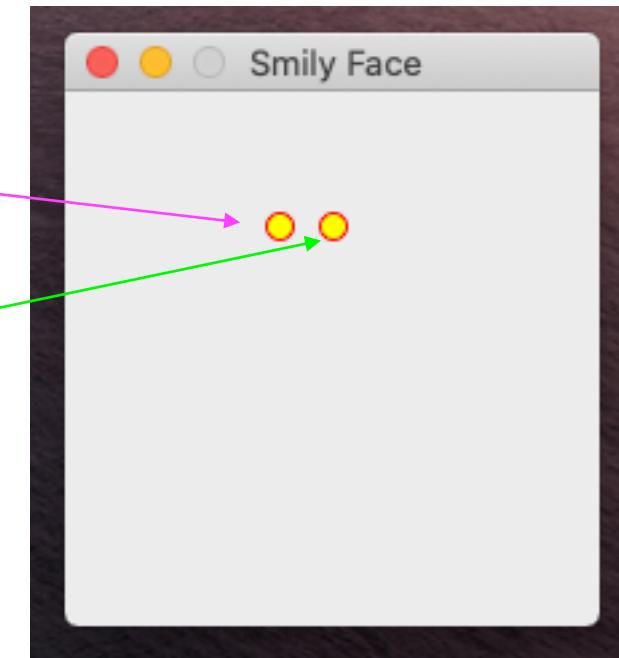
```
win = GraphWin("Smily Face")
leftEye = Circle(Point(80, 50), 5)
leftEye.setFill('yellow')
leftEye.setOutline('red')
leftEye.draw(win)
rightEye = Circle(Point(100, 50), 5)
rightEye.setFill('yellow')
rightEye.setOutline('red')
rightEye.draw(win)
```



We usually have methods that clones objects

Aliasing

```
win = GraphWin("Smily Face")
leftEye = Circle(Point(80, 50), 5)
leftEye.setFill('yellow')
leftEye.setOutline('red')
leftEye.draw(win)
rightEye = leftEye.clone()
rightEye.move(20,0)
rightEye.draw(win)
```



Interactive Graphics

Graphical interfaces can be used for input as well as output. In a GUI environment, users typically interact with their applications by clicking on buttons, choosing items from menus, and typing information into on-screen text boxes. These applications use a technique called *event-driven* programming. Basically, the program draws a set of interface elements (often called *widgets*) on the screen, and then waits for the user to do something.

Interactive Graphics

When the user moves the mouse, clicks a button, or types a key on the keyboard, this generates an event. Basically, an event is an object that encapsulates data about what just happened. The event object is then sent off to an appropriate part of the program to be processed. For example, a click on a button might produce a *button event*. This event would be passed to the button-handling code, which would then perform the appropriate action corresponding to that button.

Getting Mouse Clicks

```
# click.py
from graphics import *

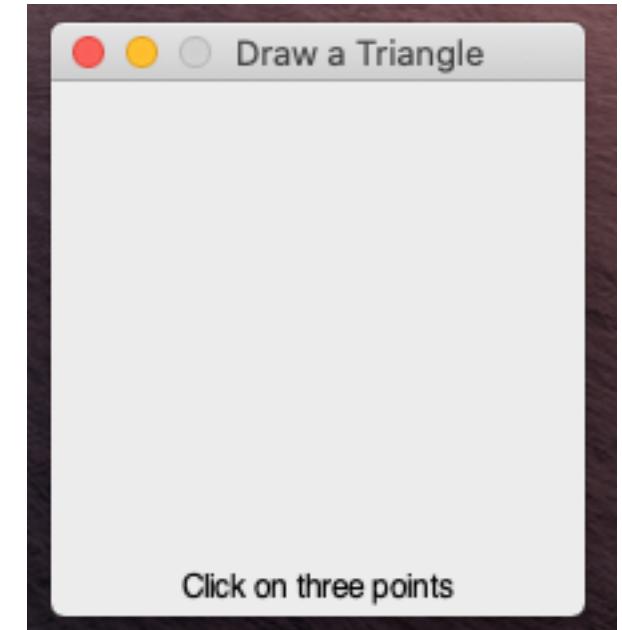
def main():
    win = GraphWin("Click Me!")
    for i in range(10):
        p = win.getMouse()
        print("You clicked at:", p.getX(), p.getY())

main()
```

Draw Triangle Using Three Mouse Clicks

First part

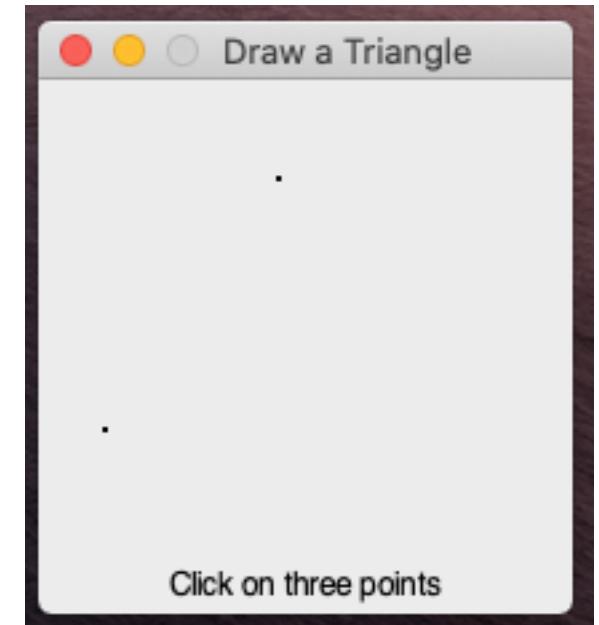
```
# draw a graphic window, setting coords :  
# Lower left (0,0), upper right (10,10)  
win = GraphWin("Draw a Triangle")  
win.setCoords(0.0, 0.0, 10.0, 10.0)  
message = Text(Point(5, 0.5), "Click on three points")  
message.draw(win)
```



Draw Triangle Using Three Mouse Clicks

Second part

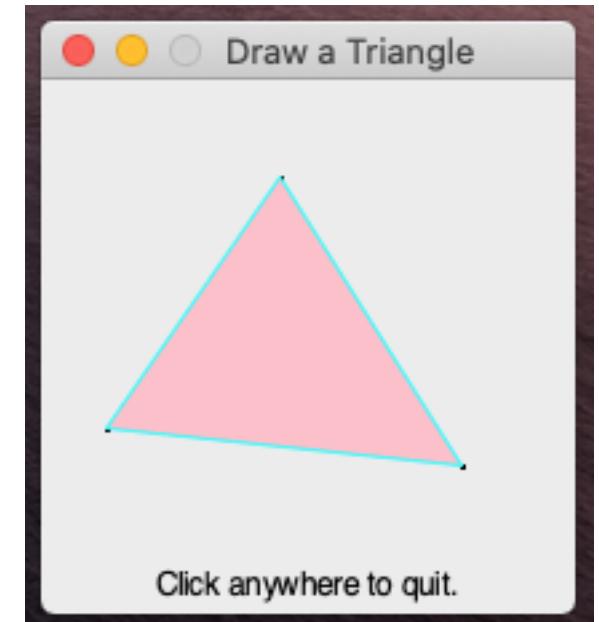
```
# Get and draw three vertices of triangle  
p1 = win.getMouse()  
p1.draw(win)  
p2 = win.getMouse()  
p2.draw(win)  
p3 = win.getMouse()  
p3.draw(win)
```



Draw Triangle Using Three Mouse Clicks

Last part

```
# Use Polygon object to draw the triangle
triangle = Polygon(p1, p2, p3)
triangle.setFill("pink")
triangle.setOutline("cyan")
triangle.draw(win)
```

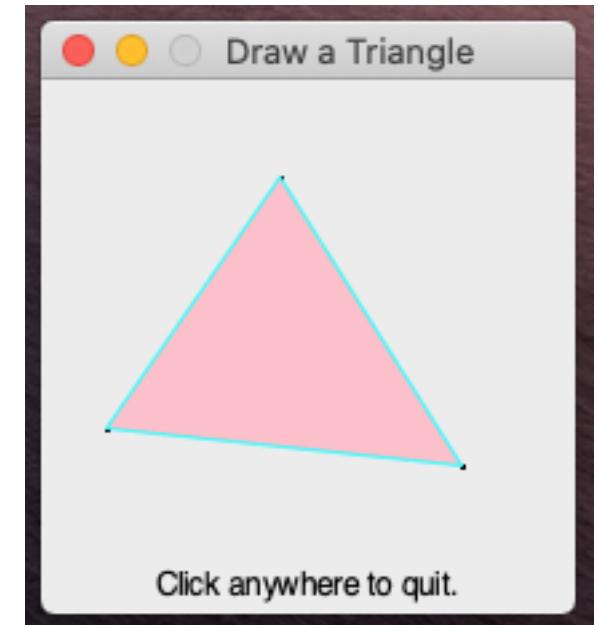


Draw Triangle Using Three Mouse Clicks

Last part

```
# Use Polygon object to draw the triangle
triangle = Polygon(p1, p2, p3)
triangle.setFill("pink")
triangle.setOutline("cyan")
triangle.draw(win)
```

```
# Wait for another click to exit
message.setText("Click anywhere to quit.")
win.getMouse()
```



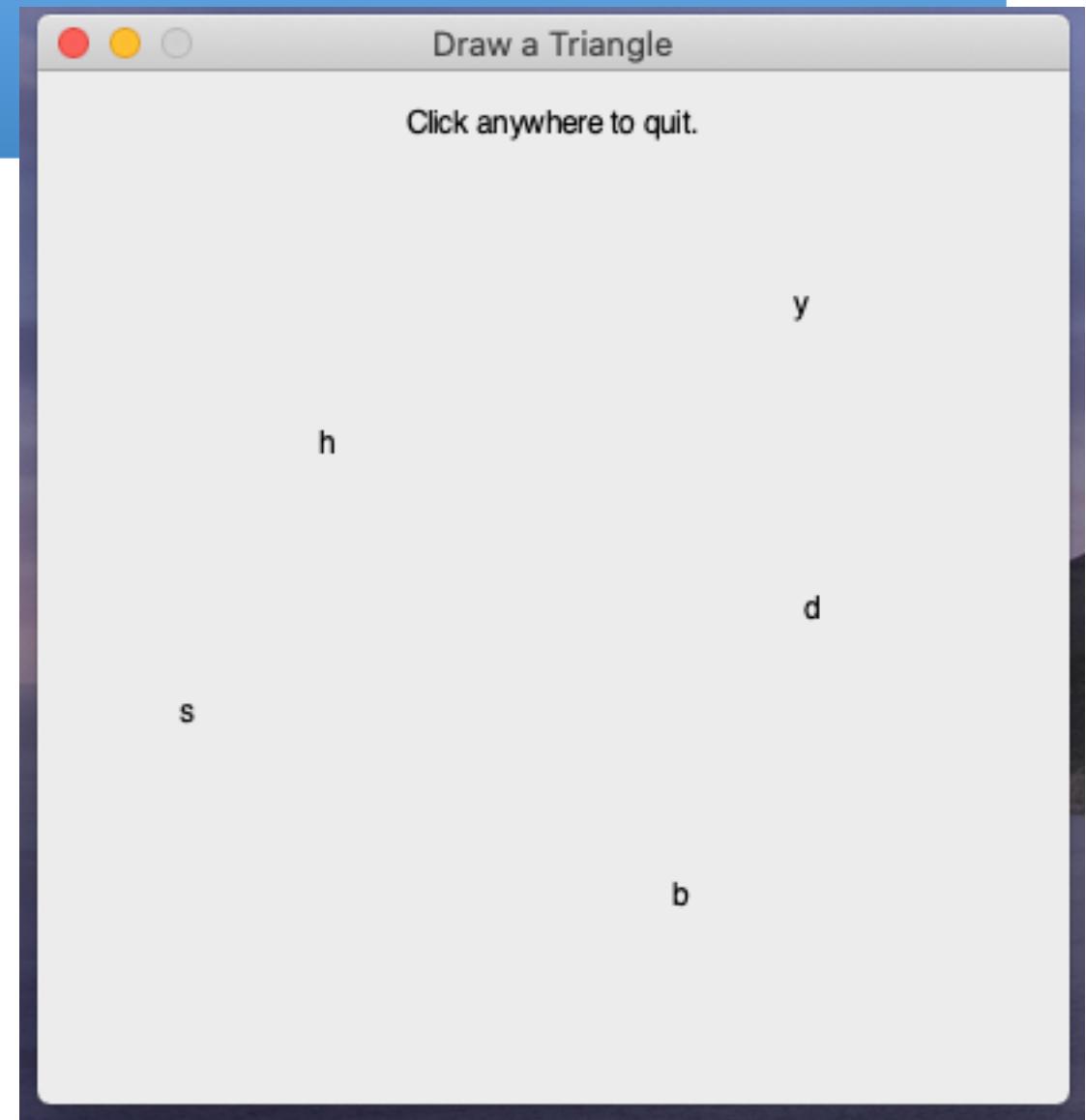
Mouse and Key events

```
def writeText():
    win = GraphWin("Draw a Triangle", 400, 400)
    win.setCoords(0.0, 0.0, 10.0, 10.0)
    message = Text(Point(5, 9.5), "Click and type")
    message.draw(win)

    for i in range(5):
        m = win.getMouse()
        k = win.getKey()
        lbl = Text(m, k)
        lbl.draw(win)

    message.setText("Click anywhere to quit.")
    win.getMouse()

if __name__ == '__main__':
    writeText()
```



Example: Celsius to Fahrenheit Conversion

```
win = GraphWin("Celsius Converter", 400, 400)
win.setCoords(0.0, 0.0, 4.0, 4.0)

# Draw the interface
Text(Point(1, 3), " Celsius: ").draw(win)
Text(Point(1, 1), " Fahrenheit : ").draw(win)
inputText = Entry(Point(2.5, 3), 5)
inputText.setText("0.0")
inputText.draw(win)

outputText = Text(Point(2.5, 1), "?")
outputText.draw(win)

button = Text(Point(2.5, 2.0), "Press mouse to convert")
button.draw(win)
```

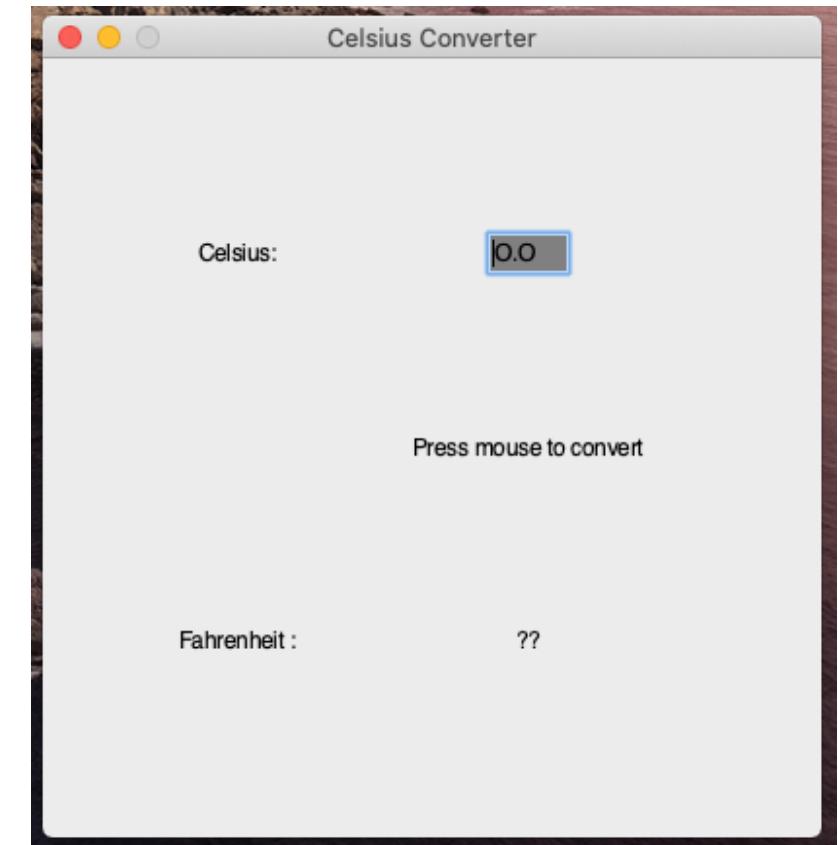
Mouse and Key events

```
win = GraphWin("Celsius Converter", 400, 400)
win.setCoords(0.0, 0.0, 4.0, 4.0)

# Draw the interface
Text(Point(1, 3), " Celsius: ").draw(win)
Text(Point(1, 1), " Fahrenheit : ").draw(win)
inputText = Entry(Point(2.5, 3), 5)
inputText.setText("0.0")
inputText.draw(win)

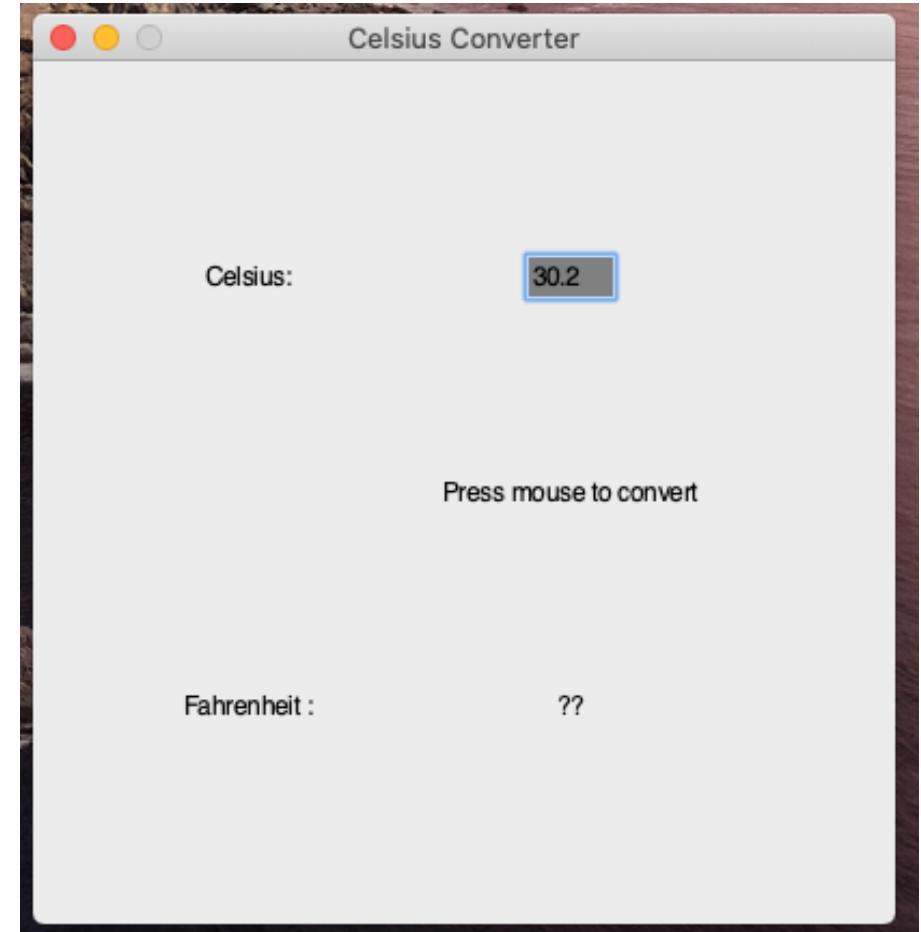
outputText = Text(Point(2.5, 1), "?")
outputText.draw(win)

button = Text(Point(2.5, 2.0), "Press mouse to convert")
button.draw(win)
```



Mouse and Key events

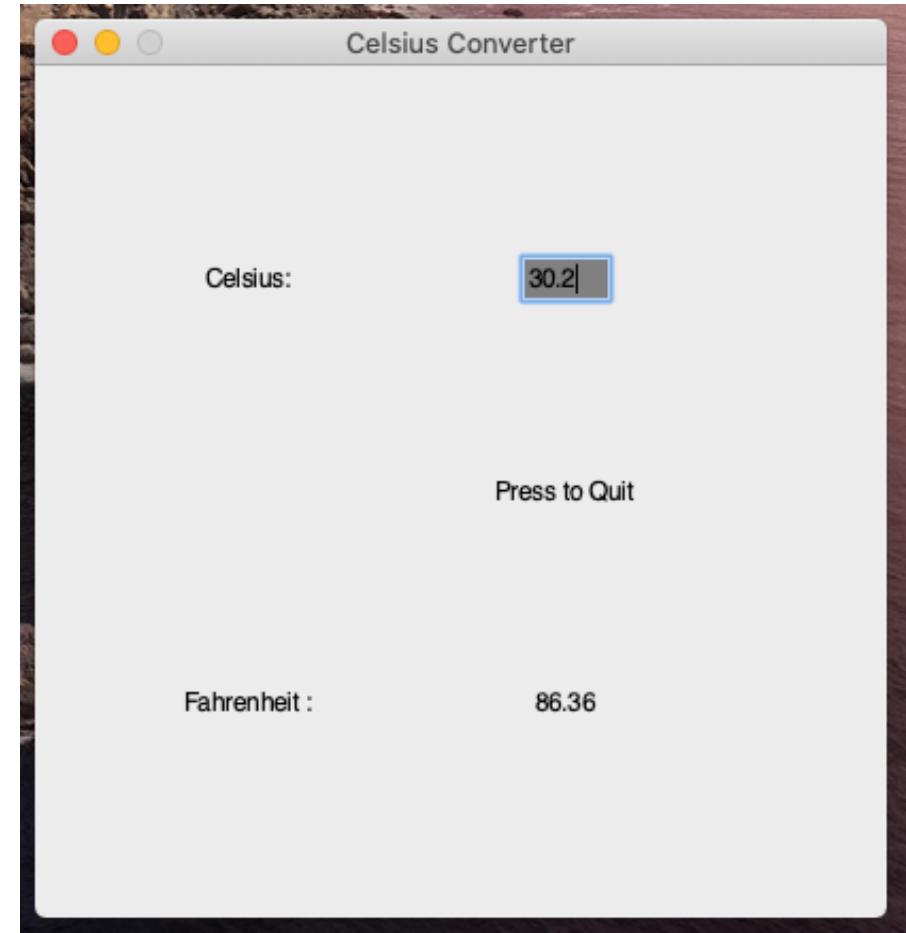
```
# wait for a mouse click  
win.getMouse()  
  
# convert input  
celsius = float(inputText.getText())  
fahrenheit = 9.0 / 5.0 * celsius + 32
```



Mouse and Key events

```
# wait for a mouse click  
win.getMouse()  
  
# convert input  
celsius = float(inputText.getText())  
fahrenheit = 9.0 / 5.0 * celsius + 32
```

```
# display output and change button  
outputText.setText(round(fahrenheit, 2))  
button.setText(" Press to Quit ")
```

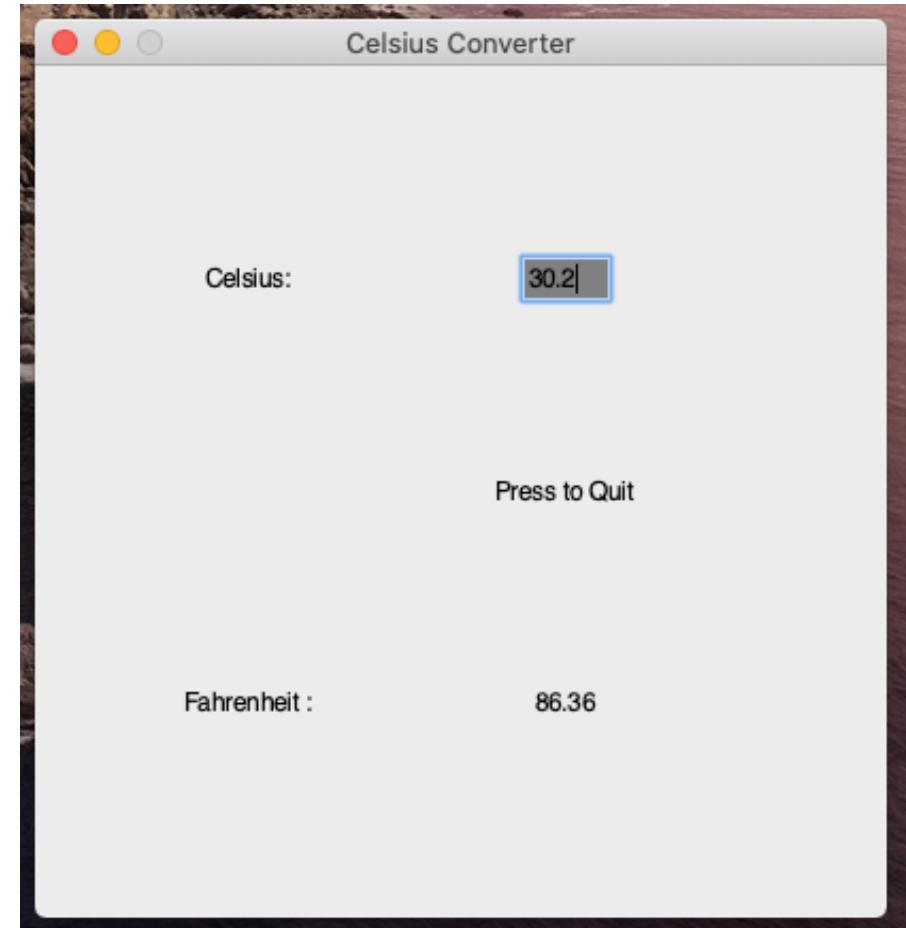


Mouse and Key events

```
# wait for a mouse click  
win.getMouse()  
  
# convert input  
celsius = float(inputText.getText())  
fahrenheit = 9.0 / 5.0 * celsius + 32
```

```
# display output and change button  
outputText.setText(round(fahrenheit, 2))  
button.setText(" Press to Quit ")
```

```
# wait for click and then quit  
win.getMouse()  
win.close()
```



Summary

An object is a computational entity that combines data and operations. Objects know stuff and can do stuff. An object's data is stored in instance variables, and its operations are called methods.

Summary

Every object is an instance of some class. It is the class that determines what methods an object will have. An instance is created by calling a constructor method.

Summary

An object's attributes are accessed via dot notation. Generally computations with objects are performed by calling on an object's methods. Accessor methods return information about the instance variables of an object. Mutator methods change the value(s) of instance variables.

Summary

The graphics module supplied with this book provides a number of classes that are useful for graphics programming. A `GraphWin` is an object that represents a window on the screen for displaying graphics. Various graphical objects such as `Point`, `Line`, `Circle`, `Rectangle`, `Oval`, `Polygon`, and `Text` may be drawn in a `GraphWin`. Users may interact with a `GraphWin` by clicking the mouse or typing into an `Entry` box.

Summary

An important consideration in graphical programming is the choice of an appropriate coordinate system. The graphics library provides a way of automating certain coordinate transformations.

Summary

The situation where two variables refer to the same object is called aliasing. Aliasing can sometimes cause unexpected results. Use of the `clone` method in the graphics library can help prevent these situations.

Lecture 17

ADT and Classes 2

Dr. Hacer Yalım Keleş

hkeles@ankara.edu.tr

Disclaimer: Content of this lecture slides are mostly from **Chapter 8.1** of Guttag's Book: "*Introduction to Computation and Programming Using Python*".

Example Class: Person

RECAP

```
class Person:

    def __init__(self, name, surname, address, telephone, email):
        self.name = name
        self.surname = surname
        self.address = address
        self.telephone = telephone
        self.email = email

    def print_info(self):
        print(self.name, self.surname)
        print(self.address)
        print("contact:", self.email)
```

Example Class: Person

RECAP

```
class Person:

    def __init__(self, name, surname, address, telephone, email):
        self.name = name
        self.surname = surname
        self.address = address
        self.telephone = telephone
        self.email = email

    def print_info(self):
        print(self.name, self.surname)
        print(self.address)
        print("contact:", self.email)
```

```
#create a person object
John = Person(
    "John",
    "Simith",
    "Ata Street, Ankara",
    "555 444 00 01",
    "john.simith@blabla.com"
)
#call object method
John.print_info()
```

Example Class: Person

RECAP

```
class Person:

    def __init__(self, name, surname, address, telephone, email):
        self.name = name
        self.surname = surname
        self.address = address
        self.telephone = telephone
        self.email = email

    def print_info(self):
        print(self.name, self.surname)
        print(self.address)
        print("contact:", self.email)
```

```
#create a person object
John = Person(
    "John",
    "Simith",
    "Ata Street, Ankara",
    "555 444 00 01",
    "john.simith@blabla.com"
)
#call object method
John.print_info()
```

John Simith
Ata Street, Ankara
contact: john.simith@blabla.com

Example Class: Person

RECAP

```
class Person:

    def __init__(self, name, surname, address, telephone, email):
        self.name = name
        self.surname = surname
        self.address = address
        self.telephone = telephone
        self.email = email

    def print_info(self):
        print(self.name, self.surname)
        print(self.address)
        print("contact:", self.email)
```

```
#create a person object
John = Person(
    "John",
    "Simith",
    "Ata Street, Ankara",
    "555 444 00 01",
    "john.simith@blabla.com"
)
#call object method
John.print_info()
```

```
print(John.name)
print(John.surname)
```

John
Simith

Example Class: Person

RECAP

```
class Person:

    def __init__(self, name, surname, address, telephone, email):
        self.name = name
        self.surname = surname
        self.address = address
        self.telephone = telephone
        self.email = email

    def print_info(self):
        print(self.name, self.surname)
        print(self.address)
        print("contact:", self.email)
```

```
#create a person object
John = Person(
    "John",
    "Simith",
    "Ata Street, Ankara",
    "555 444 00 01",
    "john.simith@blabla.com"
)
#call object method
John.print_info()
```

```
print(John.name)
print(John.surname)
```

John
Simith

```
<__main__.Person object at 0x7feac9e90f40>
<class '__main__.Person'>
<class 'method'>
```

```
print(John)
print(type(John))
print(type(John.print_info))
```

Example Class: Person

RECAP

```
class Person:  
    ...
```

Definition of Person class

```
def __init__(self, ... ):  
    ...
```

Object initializer.
self: current object - instance

```
def print_info(self):  
    ...
```

Object method - member function
self: current object - instance

```
John = Person(...)
```

An instance of Person class

```
John.name = "Johnny"
```

A member variable of Person object named John, set to "Johnny"
State of the object John has been modified.

Example: IntSet class

```
class IntSet(object):
    """An intSet is a set of integers"""
    #Information about the implementation (not the abstraction)
    #The value of the set is represented by a list of ints, self.vals.
    #Each int in the set occurs in self.vals exactly once.

    def __init__(self):
        """Create an empty set of integers"""
        self.vals = []
```

Example: IntSet class

```
class IntSet(object):
    """An intSet is a set of integers"""
    #Information about the implementation (not the abstraction)
    #The value of the set is represented by a list of ints, self.vals.
    #Each int in the set occurs in self.vals exactly once.

    def __init__(self):
        """Create an empty set of integers"""
        self.vals = []
```

representation invariant:
provides valid representations of
class instances
IntSet: vals contains no duplicates

Example: IntSet class

```
class IntSet(object):
    """An intSet is a set of integers"""
    #Information about the implementation (not the abstraction)
    #The value of the set is represented by a list of ints, self.vals.
    #Each int in the set occurs in self.vals exactly once.

    def __init__(self):
        """Create an empty set of integers"""
        self.vals = []
```

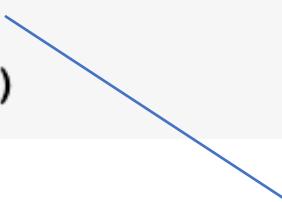


instance variable
Each instance has its own (different) vals member.

```
class IntSet(object):
    """An intSet is a set of integers"""
    #Information about the implementation (not the abstraction)
    #The value of the set is represented by a list of ints, self.vals.
    #Each int in the set occurs in self.vals exactly once.

    def __init__(self):
        """Create an empty set of integers"""
        self.vals = []

    def insert(self, e):
        """Assumes e is an integer and inserts e into self"""
        if not e in self.vals:
            self.vals.append(e)
```



to satisfy representation invariant: no duplicates

```
class IntSet(object):
    """An intSet is a set of integers"""
    #Information about the implementation (not the abstraction)
    #The value of the set is represented by a list of ints, self.vals.
    #Each int in the set occurs in self.vals exactly once.

    def __init__(self):
        """Create an empty set of integers"""
        self.vals = []

    def insert(self, e):
        """Assumes e is an integer and inserts e into self"""
        if not e in self.vals:
            self.vals.append(e)

    def member(self, e):
        """Assumes e is an integer
        Returns True if e is in self, and False otherwise"""
        return e in self.vals
```

```
def __init__(self):
    """Create an empty set of integers"""
    self.vals = []

def insert(self, e):
    """Assumes e is an integer and inserts e into self"""
    if not e in self.vals:
        self.vals.append(e)

def member(self, e):
    """Assumes e is an integer
    Returns True if e is in self, and False otherwise"""
    return e in self.vals

def remove(self, e):
    """Assumes e is an integer and removes e from self
    Raises ValueError if e is not in self"""
    try:
        self.vals.remove(e)
    except:
        raise ValueError(str(e) + ' not found')
```

```
def member(self, e):
    """Assumes e is an integer
    Returns True if e is in self, and False otherwise"""
    return e in self.vals

def remove(self, e):
    """Assumes e is an integer and removes e from self
    Raises ValueError if e is not in self"""
    try:
        self.vals.remove(e)
    except:
        raise ValueError(str(e) + ' not found')

def getMembers(self):
    """Returns a list containing the elements of self.
    Nothing can be assumed about the order of the elements"""
    return self.vals[:]
```

```
def remove(self, e):
    """Assumes e is an integer and removes e from self
    Raises ValueError if e is not in self"""
    try:
        self.vals.remove(e)
    except:
        raise ValueError(str(e) + ' not found')

def getMembers(self):
    """Returns a list containing the elements of self.
    Nothing can be assumed about the order of the elements"""
    return self.vals[:]

def __str__(self):
    """Returns a string representation of self"""
    self.vals.sort()
    result = ''
    for e in self.vals:
        result = result + str(e) + ','
    return '{' + result[:-1] + '}' # -1 omits trailing comma
```

```
class IntSet(object):

    def __init__(self):
        """Create an empty set of integers"""
        self.vals = []

    def insert(self, e):
        """Assumes e is an integer and inserts e into self"""
        if not e in self.vals:
            self.vals.append(e)

    def member(self, e):
        """Assumes e is an integer
        Returns True if e is in self, and False otherwise"""
        return e in self.vals

    def remove(self, e):
        """Assumes e is an integer and removes e from self
        Raises ValueError if e is not in self"""
        try:
            self.vals.remove(e)
        except:
            raise ValueError(str(e) + ' not found')

    def getMembers(self):
        """Returns a list containing the elements of self.
        Nothing can be assumed about the order of the elements"""
        return self.vals[:]

    def __str__(self):
        """Returns a string representation of self"""
        self.vals.sort()
        result = ''
        for e in self.vals:
            result = result + str(e) + ','
        return '{' + result[:-1] + '}' # -1 omits trailing comma
```

→ **vals** is the data attribute of the class instances
that are created from IntSet class.

```

class IntSet(object):

    def __init__(self):
        """Create an empty set of integers"""
        self.vals = []

    def insert(self, e):
        """Assumes e is an integer and inserts e into self"""
        if not e in self.vals:
            self.vals.append(e)

    def member(self, e):
        """Assumes e is an integer
        Returns True if e is in self, and False otherwise"""
        return e in self.vals

    def remove(self, e):
        """Assumes e is an integer and removes e from self
        Raises ValueError if e is not in self"""
        try:
            self.vals.remove(e)
        except:
            raise ValueError(str(e) + ' not found')

    def getMembers(self):
        """Returns a list containing the elements of self.
        Nothing can be assumed about the order of the elements"""
        return self.vals[:]

    def __str__(self):
        """Returns a string representation of self"""
        self.vals.sort()
        result = ''
        for e in self.vals:
            result = result + str(e) + ','
        return '{' + result[:-1] + '}' # -1 omits trailing comma

```

~~s = IntSet() #creates a new object of type IntSet
s.insert(3)
s.insert(5)
print(s.member(3))~~

True

- an instance object of type class IntSet
- IntSet.__init__ function is called
- *s* -the newly created instance object- is passed to bind to formal parameter *self*.

```

class IntSet(object):

    def __init__(self):
        """Create an empty set of integers"""
        self.vals = []

    def insert(self, e):
        """Assumes e is an integer and inserts e into self"""
        if not e in self.vals:
            self.vals.append(e)

    def member(self, e):
        """Assumes e is an integer
        Returns True if e is in self, and False otherwise"""
        return e in self.vals

    def remove(self, e):
        """Assumes e is an integer and removes e from self
        Raises ValueError if e is not in self"""
        try:
            self.vals.remove(e)
        except:
            raise ValueError(str(e) + ' not found')

    def getMembers(self):
        """Returns a list containing the elements of self.
        Nothing can be assumed about the order of the elements"""
        return self.vals[:]

    def __str__(self):
        """Returns a string representation of self"""
        self.vals.sort()
        result = ''
        for e in self.vals:
            result = result + str(e) + ','
        return '{' + result[:-1] + '}' # -1 omits trailing comma

```

```

s = IntSet() #creates a new object of type IntSet
s.insert(3)
s.insert(5)
print(s.member(3))

```

True

```
print(s)
```

{3,5}

```

class IntSet(object):

    def __init__(self):
        """Create an empty set of integers"""
        self.vals = []

    def insert(self, e):
        """Assumes e is an integer and inserts e into self"""
        if not e in self.vals:
            self.vals.append(e)

    def member(self, e):
        """Assumes e is an integer
        Returns True if e is in self, and False otherwise"""
        return e in self.vals

    def remove(self, e):
        """Assumes e is an integer and removes e from self
        Raises ValueError if e is not in self"""
        try:
            self.vals.remove(e)
        except:
            raise ValueError(str(e) + ' not found')

    def getMembers(self):
        """Returns a list containing the elements of self.
        Nothing can be assumed about the order of the elements"""
        return self.vals[:]

    def __str__(self):
        """Returns a string representation of self"""
        self.vals.sort()
        result = ''
        for e in self.vals:
            result = result + str(e) + ','
        return '{' + result[:-1] + '}' # -1 omits trailing comma

```

```

s = IntSet() #creates a new object of type IntSet
s.insert(3)
s.insert(5)
print(s.member(3))

```

True

```
print(s)
```

{3,5}

```
s.insert(3)
print(s)
```

{3,5}

```

class IntSet(object):

    def __init__(self):
        """Create an empty set of integers"""
        self.vals = []

    def insert(self, e):
        """Assumes e is an integer and inserts e into self"""
        if not e in self.vals:
            self.vals.append(e)

    def member(self, e):
        """Assumes e is an integer
        Returns True if e is in self, and False otherwise"""
        return e in self.vals

    def remove(self, e):
        """Assumes e is an integer and removes e from self
        Raises ValueError if e is not in self"""
        try:
            self.vals.remove(e)
        except:
            raise ValueError(str(e) + ' not found')

    def getMembers(self):
        """Returns a list containing the elements of self.
        Nothing can be assumed about the order of the elements"""
        return self.vals[:]

    def __str__(self):
        """Returns a string representation of self"""
        self.vals.sort()
        result = ''
        for e in self.vals:
            result = result + str(e) + ','
        return '{' + result[:-1] + '}' # -1 omits trailing comma

```

```

s = IntSet() #creates a new object of type IntSet
s.insert(3)
s.insert(5)
print(s.member(3))

```

True

```
print(s)
```

{3,5}

```
s.insert(3)
print(s)
```

{3,5}

```
s.remove(3)
print(s)
```

{5}

```

class IntSet(object):

    def __init__(self):
        """Create an empty set of integers"""
        self.vals = []

    def insert(self, e):
        """Assumes e is an integer and inserts e into self"""
        if not e in self.vals:
            self.vals.append(e)

    def member(self, e):
        """Assumes e is an integer
        Returns True if e is in self, and False otherwise"""
        return e in self.vals

    def remove(self, e):
        """Assumes e is an integer and removes e from self
        Raises ValueError if e is not in self"""
        try:
            self.vals.remove(e)
        except:
            raise ValueError(str(e) + ' not found')

    def getMembers(self):
        """Returns a list containing the elements of self.
        Nothing can be assumed about the order of the elements"""
        return self.vals[:]

    def __str__(self):
        """Returns a string representation of self"""
        self.vals.sort()
        result = ''
        for e in self.vals:
            result = result + str(e) + ','
        return '{' + result[:-1] + '}' # -1 omits trailing comma

```

```

s = IntSet() #creates a new object of type IntSet
s.insert(3)
s.insert(5)
print(s.member(3))

```

True

```
print(s)
```

{3,5}

```
s.insert(3)
print(s)
```

{3,5}

```
s.remove(3)
print(s)
```

{5}

```
s.insert(8)
a = s.getMembers()
print(a)
```

[5, 8]

Example: Student-Faculty-Staff

Each student would have a family name, a given name, a home address, a year, some grades, etc.

Is there an abstraction that covers the common attributes of students, faculty members, and staff?

Example: Person Class

```
import datetime
```

```
class Person():
    def __init__(self, name):
        """Create a person"""
        self.name = name
        try:
            lastBlank = name.rindex(' ')
            self.lastName = name[lastBlank+1:]
            self.name = name[:lastBlank]
        except:
            self.lastName = name
        self.birthday = None

    def getName(self):
        """Returns self's full name"""
        return self.name

    def getLastNames(self):
        """Returns self's last name"""
        return self.lastName
```

```
def setBirthday(self, birthdate):
    """Assumes birthdate is of type datetime.date
    Sets self's birthday to birthdate"""
    self.birthday = birthdate

def getAge(self):
    """Returns self's current age in days"""
    if self.birthday == None:
        raise ValueError
    return (datetime.date.today() - self.birthday).days

def __lt__(self, other):
    """Returns True if self's name is lexicographically
    less than other's name, and False otherwise"""
    if self.lastName == other.lastName:
        return self.name < other.name
    return self.lastName < other.lastName

def __str__(self):
    """Returns self's name"""
    return self.name + " " + self.lastName
```

Example: Person Class

```
import datetime
```

```
class Person():
    def __init__(self, name):
        """Create a person"""
        self.name = name
        try:
            lastBlank = name.rindex(' ')
            self.lastName = name[lastBlank+1:]
            self.name = name[:lastBlank]
        except:
            self.lastName = name
        self.birthday = None

    def getName(self):
        """Returns self's full name"""
        return self.name

    def getLastName(self):
        """Returns self's last name"""
        return self.lastName
```

```
me = Person('Michael Guttag')
him = Person('Barack Hussein Obama')
her = Person('Madonna')
print(me.getName(), "\t\t", me.getLastName())
print(him.getName(), "\t\t", him.getLastName())
print(her.getName(), "\t\t", her.getLastName())
```

Michael
Barack Hussein
Madonna

Guttag
Obama
Madonna

Example: Person Class

```
def setBirthday(self, birthdate):
    """Assumes birthdate is of type datetime.date
    Sets self's birthday to birthdate"""
    self.birthday = birthdate

def getAge(self):
    """Returns self's current age in days"""
    if self.birthday == None:
        raise ValueError
    return (datetime.date.today() - self.birthday).days

def __lt__(self, other):
    """Returns True if self's name is lexicographically
    less than other's name, and False otherwise"""
    if self.lastName == other.lastName:
        return self.name < other.name
    return self.lastName < other.lastName

def __str__(self):
    """Returns self's name"""
    return self.name + " " + self.lastName
```

→ Overloads < operator, to compare Person objects

```
print(me<him)
```

True

Example: Person Class

You can also use this class with methods that requires < operator, i.e. sorting.

```
def setBirthday(self, birthdate):
    """Assumes birthdate is of type datetime.date
    Sets self's birthday to birthdate"""
    self.birthday = birthdate

def getAge(self):
    """Returns self's current age in days"""
    if self.birthday == None:
        raise ValueError
    return (datetime.date.today() - self.birthday).days

def __lt__(self, other):
    """Returns True if self's name is lexicographically
    less than other's name, and False otherwise"""
    if self.lastName == other.lastName:
        return self.name < other.name
    return self.lastName < other.lastName

def __str__(self):
    """Returns self's name"""
    return self.name + " " + self.lastName
```

```
pList = [me, him, her]
print("Original person list:")
for p in pList:
    print(p)
pList.sort()
print("-----")
print("Sorted person list:")
for p in pList:
    print(p)
```

Original person list:
Michael Guttag
Barack Hussein Obama
Madonna Madonna

Sorted person list:
Michael Guttag
Madonna Madonna
Barack Hussein Obama

NEXT LECTURE: INHERITANCE

Lecture 18

ADT and Classes 3

Dr. Hacer Yalım Keleş

hkeles@ankara.edu.tr

Disclaimer: *Most of the slide contents are from <https://python-textbook.readthedocs.io/en/1.0>*

Object vs Class Attributes

```
class Person:

    def __init__(self, name, surname, address, telephone, email):
        self.name = name
        self.surname = surname
        self.address = address
        self.telephone = telephone
        self.email = email

    def print_info(self):
        print(self.name, self.surname)
        print(self.address)
        print("contact:", self.email)
```

All the attributes in Person class are object/instance attributes

they are added to the instance when the `__init__` method is executed.

Object vs Class Attributes

```
class Person:

    def __init__(self, name, surname, address, telephone, email):
        self.name = name
        self.surname = surname
        self.address = address
        self.telephone = telephone
        self.email = email

    def print_info(self):
        print(self.name, self.surname)
        print(self.address)
        print("contact:", self.email)
```

We can also define attributes which are set on the *class*.

These attributes will be shared by all instances of that class.

Object vs Class Attributes

We define class attributes in the body of a class, at the same indentation level as method definitions
→ one level up from the insides of methods

```
class Person:  
  
    TITLES = ('Dr', 'Mr', 'Mrs', 'Ms')  
  
    def __init__(self, title, name, surname):  
        if title not in self.TITLES:  
            raise ValueError("%s is not a valid title." % title)  
  
        self.title = title  
        self.name = name  
        self.surname = surname
```

Object vs Class Attributes

```
class Person:  
    TITLES = ('Dr', 'Mr', 'Mrs', 'Ms')  
  
    def __init__(self, title, name, surname):  
        if title not in self.TITLES:  
            raise ValueError("%s is not a valid title." % title)  
  
        self.title = title  
        self.name = name  
        self.surname = surname
```

we access the class attribute `TITLES` just like we would access an instance attribute

Object vs Class Attributes

All the `Person` objects we create will share the same `TITLES` class attribute.

```
class Person:  
  
    TITLES = ('Dr', 'Mr', 'Mrs', 'Ms')  
  
    def __init__(self, title, name, surname):  
        if title not in self.TITLES:  
            raise ValueError("%s is not a valid title." % title)  
  
        self.title = title  
        self.name = name  
        self.surname = surname
```

we access the class attribute `TITLES` just
like we would access an instance attribute

Object vs Class Attributes

Class attributes are often used to define constants which are closely associated with a particular class.

we can use class attributes from class objects, without creating an instance:

```
# we can access a class attribute from an instance  
person.TITLES  
  
# but we can also access it from the class  
Person.TITLES
```

Accessing from object instance

Accessing from class objects

Object vs Class Attributes

Note that the class object doesn't have access to any *instance* attributes
—> those are only created when an instance is created!

```
# This will give us an error
Person.name
Person.surname
```

Object vs Class Attributes

Note that the class object doesn't have access to any *instance* attributes
—> those are only created when an instance is created!

```
# This will give us an error
Person.name
Person.surname
```

When we set an attribute on an instance which has the same name as a class attribute, we are *overriding* the class attribute with an instance attribute, which will take precedence over it.

Object vs Class Attributes

We should, however, be careful when a class attribute is of a mutable type .

Remember that all instances share the same class attributes:

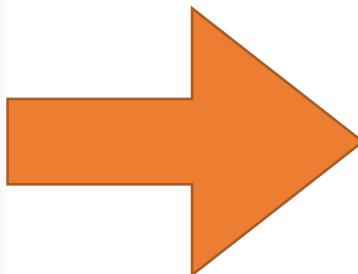
```
class Person:  
    pets = []  
  
    def add_pet(self, pet):  
        self.pets.append(pet)  
  
jane = Person()  
bob = Person()  
  
jane.add_pet("cat")  
print(jane.pets)  
print(bob.pets) # oops!
```

Object vs Class Attributes

We should, however, be careful when a class attribute is of a mutable type .

Remember that all instances share the same class attributes:

```
class Person:  
    pets = []  
  
    def add_pet(self, pet):  
        self.pets.append(pet)  
  
jane = Person()  
bob = Person()  
  
jane.add_pet("cat")  
print(jane.pets)  
print(bob.pets) # oops!
```



```
class Person:  
  
    def __init__(self):  
        self.pets = []  
  
    def add_pet(self, pet):  
        self.pets.append(pet)  
  
jane = Person()  
bob = Person()  
  
jane.add_pet("cat")  
print(jane.pets)  
print(bob.pets)
```

Create instance attribute instead

Inspecting an Object

We can check what properties are defined on an object using the `dir` function:

```
class Person:  
    def __init__(self, name, surname):  
        self.name = name  
        self.surname = surname  
  
    def fullname(self):  
        return "%s %s" % (self.name, self.surname)  
  
jane = Person("Jane", "Smith")  
  
print(dir(jane))
```

Inspecting an Object

We can check what properties are defined on an object using the `dir` function:

```
class Person:
    def __init__(self, name, surname):
        self.name = name
        self.surname = surname

    def fullname(self):
        return "%s %s" % (self.name, self.surname)

jane = Person("Jane", "Smith")

print(dir(jane))
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'fullname', 'name', 'surname']
```

Inspecting an Object

Any class that you define (in Python 3) has `object` as its parent class even if you don't explicitly say so – so your class will have a lot of default attributes and methods that any Python object has.

```
class Person:  
    def __init__(self, name, surname):  
        self.name = name  
        self.surname = surname  
  
    def fullname(self):  
        return "%s %s" % (self.name, self.surname)  
  
jane = Person("Jane", "Smith")  
  
print(dir(jane))
```

Inspecting an Object

Any class that you define (in Python 3) has `object` as its parent class even if you don't explicitly say so – so your class will have a lot of default attributes and methods that any Python object has.

```
class Person:  
    def __init__(self, name, surname):  
        self.name = name  
        self.surname = surname  
  
    def fullname(self):  
        return "%s %s" % (self.name, self.surname)  
  
jane = Person("Jane", "Smith")  
print(dir(jane))
```

This is why you can just leave out the `__init__` method out of your class if you don't have any initialisation to do – the default that you inherited from `object` (which does nothing) will be used instead. If you do write your own `__init__` method, it will *override* the default method.

Inspecting an Object

Any class that you define (in Python 3) has `object` as its parent class even if you don't explicitly say so – so your class will have a lot of default attributes and methods that any Python object has.

Many default methods and attributes that are found in built-in Python objects have names which begin and end in double underscores, like `__init__` or `__str__`.

Inspecting an Object

Any class that you define (in Python 3) has `object` as its parent class even if you don't explicitly say so – so your class will have a lot of default attributes and methods that any Python object has.

Many default methods and attributes that are found in built-in Python objects have names which begin and end in double underscores, like `__init__` or `__str__`.

We can use `dir` on any object. You can try to use it on all kinds of objects which we have already seen before, like numbers, lists, strings and functions, to see what built-in properties these objects have in common.

Some examples of special methods

- `__init__`: the initialisation method of an object, which is called when the object is created.
- `__str__`: the string representation method of an object, which is called when you use the `str` function to convert that object to a string.
- `__class__`: an attribute which stores the the class (or type) of an object – this is what is returned when you use the `type` function on the object.
- `__eq__`: a method which determines whether this object is equal to another. There are also other methods for determining if it's not equal, less than, etc.. These methods are used in object comparisons, for example when we use the equality operator `==` to check if two objects are equal.
- `__add__` is a method which allows this object to be added to another object. There are equivalent methods for all the other arithmetic operators. Not all objects support all arithmetic operations – numbers have all of these methods defined, but other objects may only have a subset.
- `__iter__`: a method which returns an iterator over the object – we will find it on strings, lists and other iterables. It is executed when we use the `iter` function on the object.
- `__len__`: a method which calculates the length of an object – we will find it on sequences. It is executed when we use the `len` function of an object.
- `__dict__`: a dictionary which contains all the instance attributes of an object, with their names as keys. It can be useful if we want to iterate over all the attributes of an object. `__dict__` does not include any methods, class attributes or special default attributes like `__class__`.

We can override special methods

```
class Person:  
    def __init__(self, name, surname):  
        self.name = name  
        self.surname = surname  
  
    def __eq__(self, other): # does self == other?  
        return self.name == other.name and self.surname == other.surname  
  
    def __gt__(self, other): # is self > other?  
        if self.surname == other.surname:  
            return self.name > other.name  
        return self.surname > other.surname  
  
    # now we can define all the other methods in terms of the first two  
  
    def __ne__(self, other): # does self != other?  
        return not self == other # this calls self.__eq__(other)  
  
    def __le__(self, other): # is self <= other?  
        return not self > other # this calls self.__gt__(other)  
  
    def __lt__(self, other): # is self < other?  
        return not (self > other or self == other)  
  
    def __ge__(self, other): # is self >= other?  
        return not self < other
```

Lecture 19

ADT and Classes 4

Inheritance

Dr. Hacer Yalım Keleş

hkeles@ankara.edu.tr

Disclaimer: *Most of the slide contents are from <https://python-textbook.readthedocs.io/en/1.0>*

Inheritance

Inheritance is a way of arranging objects in a hierarchy from the most general to the most specific.

An object which *inherits* from another object is considered to be a *subtype* of that object.

Inheritance

Inheritance is a way of arranging objects in a hierarchy from the most general to the most specific.

An object which *inherits* from another object is considered to be a *subtype* of that object.

As we saw in the previous chapter, all objects in Python inherit from `object`.
We can say that a string, an integer or a `Person` instance *is an object* instance.

In Object Oriented Programming terminology, inheritance defined **is-a** relation:

Person **is-a** Object

Inheritance

In Object Oriented Programming terminology, inheritance defined **is-a** relation:
Person is-a Object

a class is a *subclass* or *child class* of a class from which it inherits

Person class is a subclass of Object class

the other class is its *superclass* or *parent class*

Object class is a superclass of Person class

Inheritance

In Object Oriented Programming terminology, inheritance defined **is-a** relation:
Person is-a Object

a class is a **subclass** or **child class** of a class from which it inherits

Person class is a subclass of Object class

the other class is its **superclass** or **parent class**

Object class is a superclass of Person class

We can define a hierarchy of classes with is-a relation.
We can refer to the most generic class at the base of a hierarchy as a **base class**

Inheritance

We can put all the functionality that the objects have in common in a base class, and then define one or more subclasses with their own custom functionality.

Inheritance

We can put all the functionality that the objects have in common in a base class, and then define one or more subclasses with their own custom functionality.

Inheritance is also a way of reusing existing code easily. If we already have a class which does *almost* what we want, we can create a subclass in which we partially override some of its behaviour, or perhaps add some new functionality.

Inheritance

```
class Person:  
    def __init__(self, name, surname, number):  
        self.name = name  
        self.surname = surname  
        self.number = number
```

Inheritance

```
class Person:  
    def __init__(self, name, surname, number):  
        self.name = name  
        self.surname = surname  
        self.number = number
```

```
class Student(Person):  
    UNDERGRADUATE, POSTGRADUATE = range(2)  
  
    def __init__(self, student_type, *args, **kwargs):  
        self.student_type = student_type  
        self.classes = []  
        super(Student, self).__init__(*args, **kwargs)  
  
    def enrol(self, course):  
        self.classes.append(course)
```

*args and **kwargs

```
class Student(Person):
    UNDERGRADUATE, POSTGRADUATE = range(2)

    def __init__(self, student_type, *args, **kwargs):
        self.student_type = student_type
        self.classes = []
        super(Student, self).__init__(*args, **kwargs)

    def enrol(self, course):
        self.classes.append(course)
```

Sometimes we want to pass a variable-length list of positional or keyword parameters into a function.

We can put `*` before a parameter name to indicate that it is a variable-length tuple of positional parameters

we can use `**` to indicate that a parameter is a variable-length dictionary of keyword parameters.

By convention, the parameter name we use for the tuple is `args` and the name we use for the dictionary is `kwargs`:

*args and **kwargs

```
class Student(Person):
    UNDERGRADUATE, POSTGRADUATE = range(2)

    def __init__(self, student_type, *args, **kwargs):
        self.student_type = student_type
        self.classes = []
        super(Student, self).__init__(*args, **kwargs)

    def enrol(self, course):
        self.classes.append(course)
```

We can use `*` or `**` when we are *calling* a function to *unpack* a sequence or a dictionary into a series of individual parameters:

```
def print_args(*args):
    for arg in args:
        print(arg)

def print_kwargs(**kwargs):
    for k, v in kwargs.items():
        print("%s: %s" % (k, v))
```

*args and **kwargs

```
class Student(Person):
    UNDERGRADUATE, POSTGRADUATE = range(2)

    def __init__(self, student_type, *args, **kwargs):
        self.student_type = student_type
        self.classes = []
        super(Student, self).__init__(*args, **kwargs)

    def enrol(self, course):
        self.classes.append(course)
```

We can use `*` or `**` when we are *calling* a function to *unpack* a sequence or a dictionary into a series of individual parameters:

```
def print_args(*args):
    for arg in args:
        print(arg)

def print_kwargs(**kwargs):
    for k, v in kwargs.items():
        print("%s: %s" % (k, v))
```

```
print_args("one", "two", "three")
print_args("one", "two", "three", "four")

print_kwargs(name="Jane", surname="Doe")
print_kwargs(age=10)
```

*args and **kwargs

```
class Student(Person):
    UNDERGRADUATE, POSTGRADUATE = range(2)

    def __init__(self, student_type, *args, **kwargs):
        self.student_type = student_type
        self.classes = []
        super(Student, self).__init__(*args, **kwargs)

    def enrol(self, course):
        self.classes.append(course)
```

We can use `*` or `**` when we are *calling* a function to *unpack* a sequence or a dictionary into a series of individual parameters:

```
def print_args(*args):
    for arg in args:
        print(arg)

def print_kwargs(**kwargs):
    for k, v in kwargs.items():
        print("%s: %s" % (k, v))
```

```
my_list = ["one", "two", "three"]
print_args(*my_list)

my_dict = {"name": "Jane", "surname": "Doe"}
print_kwargs(**my_dict)
```

*args and **kwargs

```
class Student(Person):
    UNDERGRADUATE, POSTGRADUATE = range(2)

    def __init__(self, student_type, *args, **kwargs):
        self.student_type = student_type
        self.classes = []
        super(Student, self).__init__(*args, **kwargs)

    def enrol(self, course):
        self.classes.append(course)
```

We can use `*` or `**` when we are *calling* a function to *unpack* a sequence or a dictionary into a series of individual parameters:

```
def print_args(*args):
    for arg in args:
        print(arg)

def print_kwargs(**kwargs):
    for k, v in kwargs.items():
        print("%s: %s" % (k, v))
```

```
def print_args(*args):
    for i in args:
        print(i)

t = (2, 4, 7, "hello")
print_args(t)
(2, 4, 7, 'hello')

a = [2, 3, 4, 5, 6, 7, "hello"]
pargs(a)
[2, 3, 4, 5, 6, 7, 'hello']
```

*args and **kwargs

```
class Student(Person):
    UNDERGRADUATE, POSTGRADUATE = range(2)

    def __init__(self, student_type, *args, **kwargs):
        self.student_type = student_type
        self.classes = []
        super(Student, self).__init__(*args, **kwargs)

    def enrol(self, course):
        self.classes.append(course)
```

We can use `*` or `**` when we are *calling* a function to *unpack* a sequence or a dictionary into a series of individual parameters:

```
def print_args(*args):
    for arg in args:
        print(arg)

def print_kwargs(**kwargs):
    for k, v in kwargs.items():
        print("%s: %s" % (k, v))
```

```
def print_args(*args):
    for i in args:
        print(i)

t = (2, 4, 7, "hello")
print_args(t)
(2, 4, 7, 'hello')

a = [2, 3, 4, 5, 6, 7, "hello"]
pargs(a)
[2, 3, 4, 5, 6, 7, 'hello']
```

```
def print_args(*args):
    for i in args:
        print(i)

t = (2, 4, 7, "hello")
print_args(*t)
2
4
7
hello
```

*args and **kwargs

```
class Student(Person):
    UNDERGRADUATE, POSTGRADUATE = range(2)

    def __init__(self, student_type, *args, **kwargs):
        self.student_type = student_type
        self.classes = []
        super(Student, self).__init__(*args, **kwargs)

    def enrol(self, course):
        self.classes.append(course)
```

We can mix ordinary parameters, `*args` and `**kwargs` in the same function definition.

`*args` and `**kwargs` must come after all the other parameters, and `**kwargs` must come after `*args`.

You cannot have more than one variable-length list parameter or more than one variable dict parameter

```
def print_everything(name, time="morning", *args, **kwargs):
    print("Good %s, %s." % (time, name))

    for arg in args:
        print(arg)

    for k, v in kwargs.items():
        print("%s: %s" % (k, v))
```

*args and **kwargs

```
class Student(Person):
    UNDERGRADUATE, POSTGRADUATE = range(2)

    def __init__(self, student_type, *args, **kwargs):
        self.student_type = student_type
        self.classes = []
        super(Student, self).__init__(*args, **kwargs)

    def enrol(self, course):
```

If we use a `*` expression when you call a function, it must come after all the positional parameters

If we use a `**` expression it must come right at the end

```
def print_everything(*args, **kwargs):
    for arg in args:
        print(arg)

    for k, v in kwargs.items():
        print("%s: %s" % (k, v))

# we can write all the parameters individually
print_everything("cat", "dog", day="Tuesday")

t = ("cat", "dog")
d = {"day": "Tuesday"}

# we can unpack a tuple and a dictionary
print_everything(*t, **d)
# or just one of them
print_everything(*t, day="Tuesday")
print_everything("cat", "dog", **d)

# we can mix * and ** with explicit parameters
print_everything("Jane", *t, **d)
print_everything("Jane", *t, time="evening", **d)
print_everything(time="evening", *t, **d)
```

*args and **kwargs

```
class Student(Person):
    UNDERGRADUATE, POSTGRADUATE = range(2)

    def __init__(self, student_type, *args, **kwargs):
        self.student_type = student_type
        self.classes = []
        super(Student, self).__init__(*args, **kwargs)

    def enrol(self, course):
```

If we use a `*` expression when you call a function, it must come after all the positional parameters

If we use a `**` expression it must come right at the end

```
def print_everything(*args, **kwargs):
    for arg in args:
        print(arg)

    for k, v in kwargs.items():
        print("%s: %s" % (k, v))

# we can write all the parameters individually
print_everything("cat", "dog", day="Tuesday")

t = ("cat", "dog")
d = {"day": "Tuesday"}

# we can unpack a tuple and a dictionary
print_everything(*t, **d)
# or just one of them
print_everything(*t, day="Tuesday")
print_everything("cat", "dog", **d)

# we can mix * and ** with explicit parameters
print_everything("Jane", *t, **d)
print_everything("Jane", *t, time="evening", **d)
print_everything(time="evening", *t, **d)
```

WHY?

```
# none of these are allowed:
print_everything(*t, "Jane", **d)
print_everything(*t, **d, time="evening")
```

*args and **kwargs

```
class Student(Person):
    UNDERGRADUATE, POSTGRADUATE = range(2)

    def __init__(self, student_type, *args, **kwargs):
        self.student_type = student_type
        self.classes = []
        super(Student, self).__init__(*args, **kwargs)

    def enrol(self, course):
```

If we use a `*` expression when you call a function, it must come after all the positional parameters

If we use a `**` expression it must come right at the end

```
def print_everything(*args, **kwargs):
    for arg in args:
        print(arg)

    for k, v in kwargs.items():
        print("%s: %s" % (k, v))

# we can write all the parameters individually
print_everything("cat", "dog", day="Tuesday")

t = ("cat", "dog")
d = {"day": "Tuesday"}

# we can unpack a tuple and a dictionary
print_everything(*t, **d)
# or just one of them
print_everything(*t, day="Tuesday")
print_everything("cat", "dog", **d)

# we can mix * and ** with explicit parameters
print_everything("Jane", *t, **d)
print_everything("Jane", *t, time="evening", **d)
print_everything(time="evening", *t, **d)
```

WHY?

```
# none of these are allowed:
print_everything(*t, "Jane", **d)
print_everything(*t, **d, time="evening")
```

If a function takes only `*args` and `**kwargs` as its parameters, it can be called with *any set of parameters*. One or both of `args` and `kwargs` can be empty, so the function will accept any combination of positional and keyword parameters, including no parameters at all.

Inheritance

```
class Person:  
    def __init__(self, name, surname, number):  
        self.name = name  
        self.surname = surname  
        self.number = number
```

The `__init__` method of the base class initialises all the instance variables that are common to all subclasses.

```
class StaffMember(Person):  
    PERMANENT, TEMPORARY = range(2)  
  
    def __init__(self, employment_type, *args, **kwargs):  
        self.employment_type = employment_type  
        super(StaffMember, self).__init__(*args, **kwargs)
```

Inheritance

```
class Person:  
    def __init__(self, name, surname, number):  
        self.name = name  
        self.surname = surname  
        self.number = number
```

In each subclass we *override* the `__init__` method so that we can use it to initialise that class's attributes

```
class StaffMember(Person):  
    PERMANENT, TEMPORARY = range(2)  
  
    def __init__(self, employment_type, *args, **kwargs):  
        self.employment_type = employment_type  
        super(StaffMember, self).__init__(*args, **kwargs)
```

Inheritance

```
class Person:  
    def __init__(self, name, surname, number):  
        self.name = name  
        self.surname = surname  
        self.number = number
```

In each subclass we *override* the `__init__` method so that we can use it to initialise that class's attributes

but we want the parent class's attributes to be initialised as well

```
class StaffMember(Person):  
    PERMANENT, TEMPORARY = range(2)  
  
    def __init__(self, employment_type, *args, **kwargs):  
        self.employment_type = employment_type  
        super(StaffMember, self).__init__(*args, **kwargs)
```

so we need to call the parent's `__init__` method from ours

in Python 3:
`super().__init__(*args, **kwargs)`

Inheritance

```
class Person:  
    def __init__(self, name, surname, number):  
        self.name = name  
        self.surname = surname  
        self.number = number
```

In each subclass we *override* the `__init__` method so that we can use it to initialise that class's attributes

but we want the parent class's attributes to be initialised as well

```
class StaffMember(Person):  
    PERMANENT, TEMPORARY = range(2)  
  
    def __init__(self, employment_type, *args, **kwargs):  
        self.employment_type = employment_type  
        super(StaffMember, self).__init__(*args, **kwargs)
```

so we need to call the parent's `__init__` method from ours

To find the right method, we use the `super` function

Returns the correct parent object

Inheritance

```
class Person:  
    def __init__(self, name, surname, number):  
        self.name = name  
        self.surname = surname  
        self.number = number
```

In each of our overridden `__init__` methods we use those of the method's parameters which are specific to our class inside the method, and then pass the remaining parameters to the parent class's `__init__` method. A common convention is to add the specific parameters for each successive subclass to the *beginning* of the parameter list, and define all the other parameters using `*args` and `**kwargs` – then the subclass doesn't need to know the details about the parent class's parameters.

```
class StaffMember(Person):  
    PERMANENT, TEMPORARY = range(2)  
  
    def __init__(self, employment_type, *args, **kwargs):  
        self.employment_type = employment_type  
        super(StaffMember, self).__init__(*args, **kwargs)
```

Inheritance

```
class Person:  
    def __init__(self, name, surname, number):  
        self.name = name  
        self.surname = surname  
        self.number = number
```

is-a

```
class StaffMember(Person):  
    PERMANENT, TEMPORARY = range(2)  
  
    def __init__(self, employment_type, *args, **kwargs):  
        self.employment_type = employment_type  
        super(StaffMember, self).__init__(*args, **kwargs)
```

is-a

```
class Lecturer(StaffMember):  
    def __init__(self, *args, **kwargs):  
        self.courses_taught = []  
        super(Lecturer, self).__init__(*args, **kwargs)  
  
    def assign_teaching(self, course):  
        self.courses_taught.append(course)
```

Inheritance

```
class Person:  
    def __init__(self, name, surname, number):  
        self.name = name  
        self.surname = surname  
        self.number = number
```

```
class StaffMember(Person):  
    PERMANENT, TEMPORARY = range(2)  
  
    def __init__(self, employment_type, *args, **kwargs):  
        self.employment_type = employment_type  
        super(StaffMember, self).__init__(*args, **kwargs)
```

```
class Lecturer(StaffMember):  
    def __init__(self, *args, **kwargs):  
        self.courses_taught = []  
        super(Lecturer, self).__init__(*args, **kwargs)  
  
    def assign_teaching(self, course):  
        self.courses_taught.append(course)
```

```
jane = Student(Student.POSTGRADUATE, "Jane", "Smith", "SMTJNX045")  
jane.enrol(a_postgrad_course)  
  
bob = Lecturer(StaffMember.PERMANENT, "Bob", "Jones", "123456789")  
bob.assign_teaching(an_undergrad_course)
```

```
class Student(Person):  
    UNDERGRADUATE, POSTGRADUATE = range(2)  
  
    def __init__(self, student_type, *args, **kwargs):  
        self.student_type = student_type  
        self.classes = []  
        super(Student, self).__init__(*args, **kwargs)  
  
    def enrol(self, course):  
        self.classes.append(course)
```

Lecture 20

ADT and Classes 5

Class Hierarchies

Dr. Hacer Yalım Keleş

hkeles@ankara.edu.tr

Disclaimer: *Most of the slide contents are from <https://www.programiz.com/python-programming/multiple-inheritance>*

Multiple Inheritance

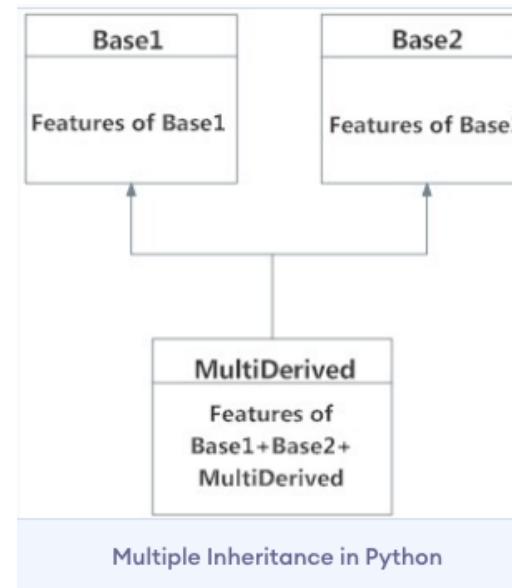
Similar to C++, a class can be derived from more than one class in Python

```
class Basel:  
    pass  
  
class Base2:  
    pass  
  
class Derived(Base1, Base2):  
    pass
```

Multiple Inheritance

Similar to C++, a class can be derived from more than one class in Python

```
class Base1:  
    pass  
  
class Base2:  
    pass  
  
class Derived(Base1, Base2):  
    pass
```

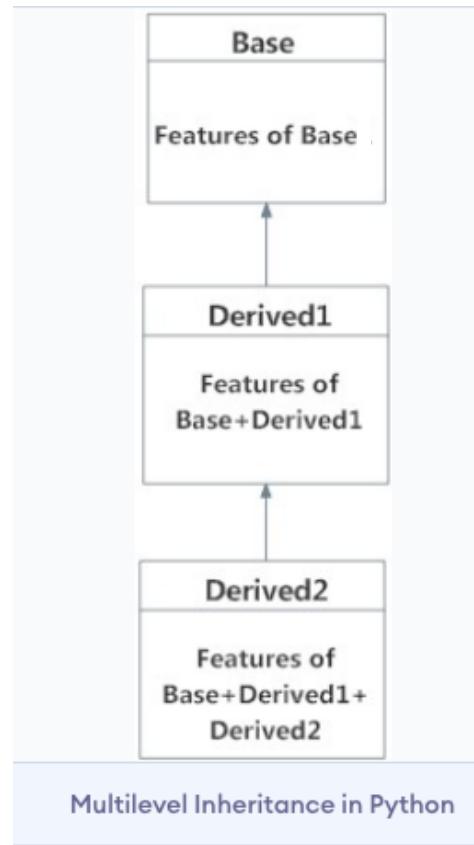


Hierarchy of Single Inheritance

Multilevel inheritance

Similar to C++, a class can be derived from more than one class in Python

```
class Base:  
    def __init__(self):  
        print("Base")  
  
class Derived1(Base):  
    def __init__(self):  
        super().__init__()  
        print("Derived1")  
  
class Derived2(Derived1):  
    def __init__(self):  
        super().__init__()  
        print("Derived2")
```

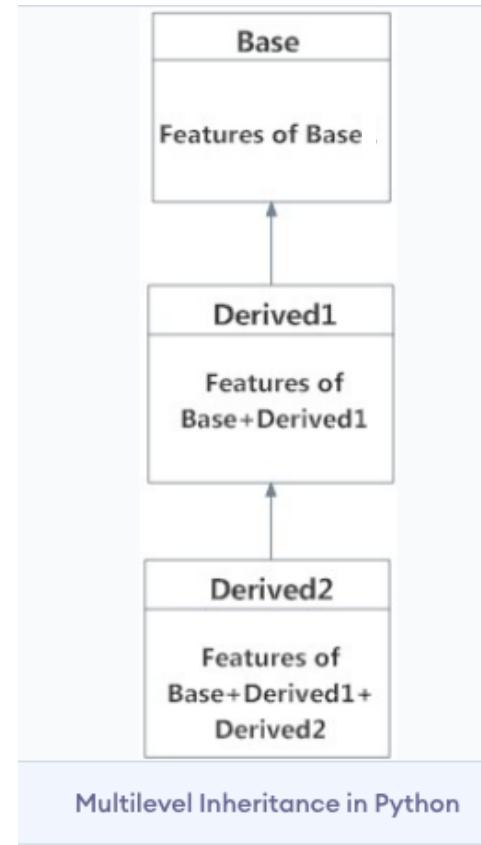


Hierarchy of Single Inheritance

Multilevel inheritance

Similar to C++, a class can be derived from more than one class in Python

```
class Base:  
    def __init__(self):  
        print("Base")  
  
class Derived1(Base):  
    def __init__(self):  
        super().__init__()  
        print("Derived1")  
  
class Derived2(Derived1):  
    def __init__(self):  
        super().__init__()  
        print("Derived2")
```



```
d = Derived2()  
  
Base  
Derived1  
Derived2
```

Method Resolution Order (MRO)

Every class in Python is derived from the `object` class. It is the most base type in Python.

All other classes, either built-in or user-defined, are derived classes and all objects are instances of the `object` class.

```
print(issubclass(list,object))
print(isinstance(5.5,object))
print(isinstance("Hello",object))
print(isinstance(d,object))
```

True
True
True
True

Derived2 class instance that we defined a minute ago..

Method Resolution Order (MRO)

In the multiple inheritance scenario, any specified attribute is searched :

- 1- in the current class; if not found,
- 2- the search continues into parent classes in depth-first, left-right fashion

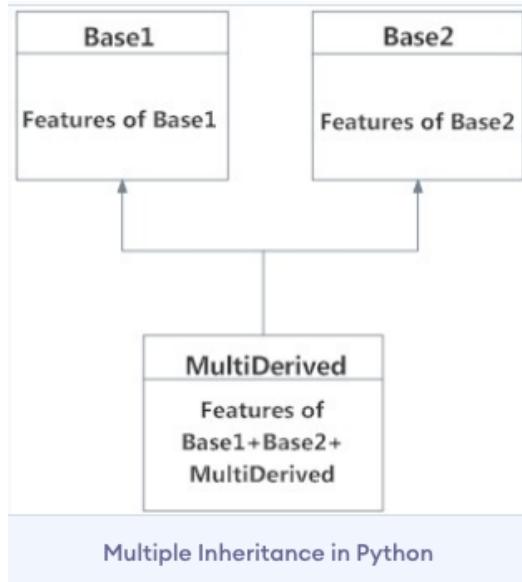
No class is searched twice.

Method Resolution Order (MRO)

In the multiple inheritance scenario, any specified attribute is searched :

- 1- in the current class; if not found,
- 2- the search continues into parent classes in depth-first, left-right fashion

No class is searched twice.



MRO: <MultiDerived, Base1, Base2, object>

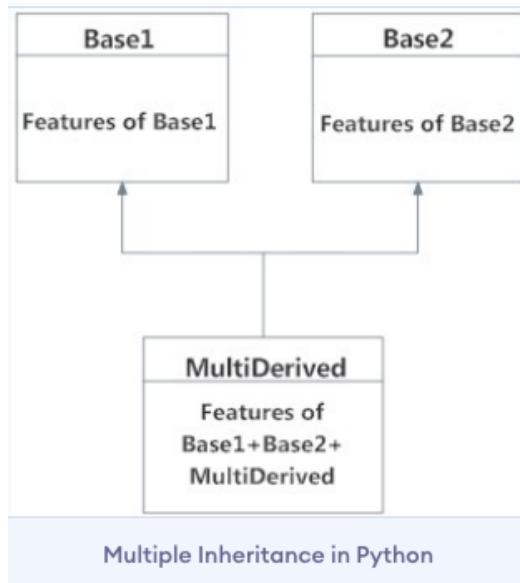
This order is also called linearization of MultiDerived class
The set of rules used to find this order is called **Method Resolution Order (MRO)**.

Method Resolution Order (MRO)

__mro__ attribute or mro() method

MRO must prevent local precedence ordering and also provide monotonicity.
It ensures that a class always appears before its parents.

In case of multiple parents, the order is the same as tuples of base classes.



MRO of a class can be viewed as the `__mro__` attribute or the `mro()` method.

`__mro__` → returns a tuple
`mro()` → returns a list

```
class Basel:
    pass

class Base2:
    pass

class MultiDerived(Base1, Base2):
    pass

print(MultiDerived.__mro__)

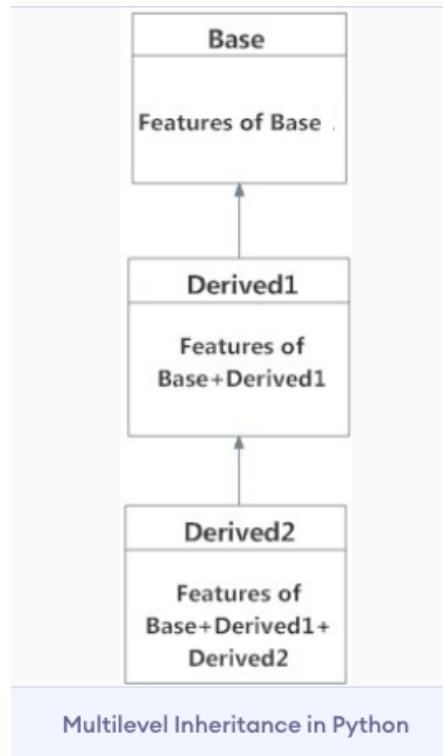
(<class '__main__.MultiDerived'>, <class '__main__.Base1'>, <class '__main__.Base2'>, <class 'object'>)
```

Method Resolution Order (MRO)

__mro__ attribute or mro() method

MRO must prevent local precedence ordering and also provide monotonicity.
It ensures that a class always appears before its parents.

In case of multiple parents, the order is the same as tuples of base classes.



MRO of a class can be viewed as the `__mro__` attribute or the `mro()` method.

`__mro__` → returns a tuple

`mro()` → returns a list

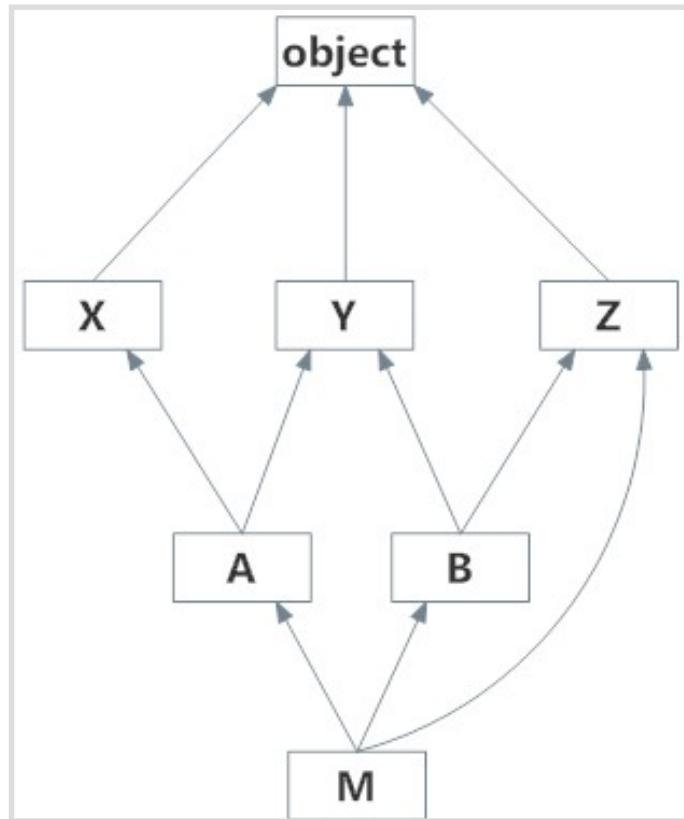
```
d = Derived2()
print(Derived2.mro())
print(Derived2.__mro__)
```

```
Base
Derived1
Derived2
[<class '__main__.Derived2'>, <class '__main__.Derived1'>, <class '__main__.Base'>, <class 'object'>
(<class '__main__.Derived2'>, <class '__main__.Derived1'>, <class '__main__.Base'>, <class 'object'>)
```

Method Resolution Order (MRO)

__mro__ attribute or mro() method

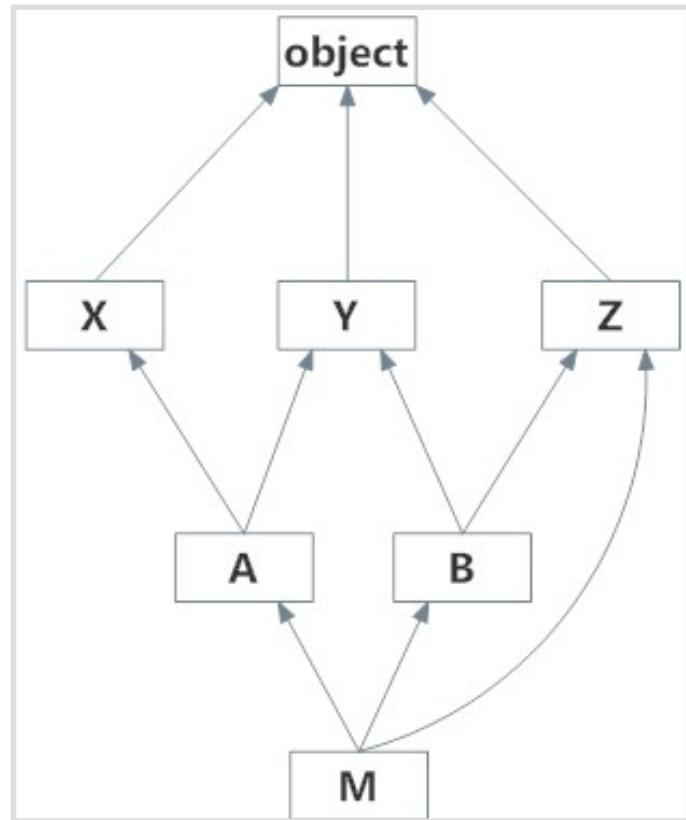
```
class X:  
    pass  
  
class Y:  
    pass  
  
class Z:  
    pass  
  
class A(X, Y):  
    pass  
  
class B(Y, Z):  
    pass  
  
class M(B, A, Z):  
    pass  
  
mrolist = M.mro()  
for s in mrolist:  
    print(s)
```



Method Resolution Order (MRO)

__mro__ attribute or mro() method

```
class X:  
    pass  
  
class Y:  
    pass  
  
class Z:  
    pass  
  
class A(X, Y):  
    pass  
  
class B(Y, Z):  
    pass  
  
class M(B, A, Z):  
    pass  
  
mrolist = M.mro()  
for s in mrolist:  
    print(s)
```



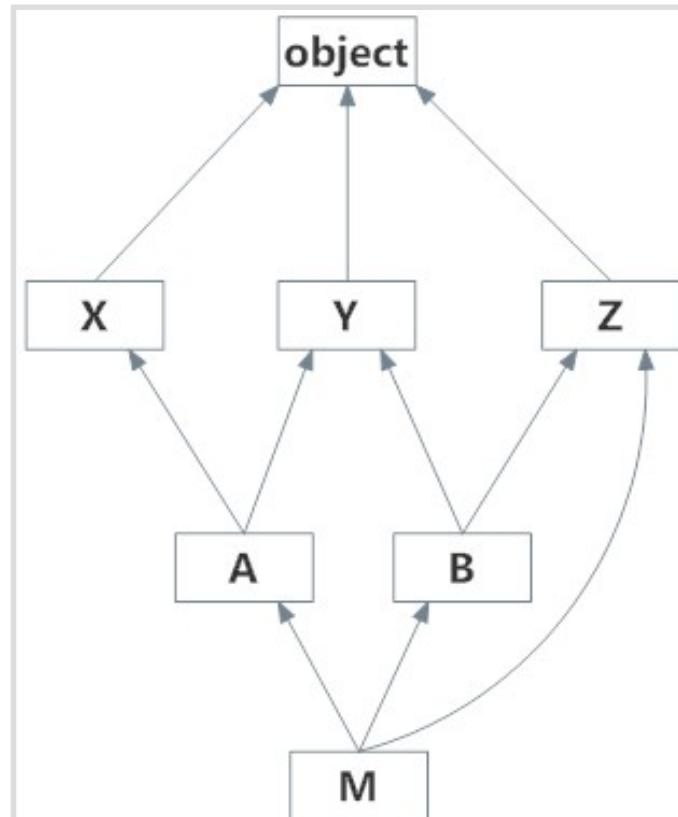
MRO must prevent local precedence ordering and also provide monotonicity. It ensures that a class always appears before its parents.

In case of multiple parents, the order is the same as tuples of base classes.

Method Resolution Order (MRO)

__mro__ attribute or mro() method

```
class X:  
    pass  
  
class Y:  
    pass  
  
class Z:  
    pass  
  
class A(X, Y):  
    pass  
  
class B(Y, Z):  
    pass  
  
class M(B, A, Z):  
    pass  
  
mrolist = M.mro()  
for s in mrolist:  
    print(s)
```



MRO must prevent local precedence ordering and also provide monotonicity. It ensures that a class always appears before its parents.

In case of multiple parents, the order is the same as tuples of base classes.

```
<class '__main__.M'>  
<class '__main__.B'>  
<class '__main__.A'>  
<class '__main__.X'>  
<class '__main__.Y'>  
<class '__main__.Z'>  
<class 'object'>
```

The Diamond Problem

The "diamond problem" is the generally used term for an ambiguity that arises when; two classes B and C inherit from a superclass A, and another class D inherits from both B and C.

If there is a method "m" in A that B or C (or even both of them) has overridden, and furthermore, if it does not override this method, then the question is which version of the method does D inherit?

It could be the one from A, B or C.

The Diamond Problem

```
class A:  
    def m(self):  
        print("m of A called")  
  
class B(A):  
    def m(self):  
        print("m of B called")  
  
class C(A):  
    def m(self):  
        print("m of C called")  
  
class D(B,C):  
    pass  
  
d = D()  
d.m()
```

The Diamond Problem

```
class A:  
    def m(self):  
        print("m of A called")  
  
class B(A):  
    def m(self):  
        print("m of B called")  
  
class C(A):  
    def m(self):  
        print("m of C called")  
  
class D(B,C):  
    pass  
  
d = D()  
d.m()
```

m of B called

The Diamond Problem

```
class A:  
    def m(self):  
        print("m of A called")  
  
class B(A):  
    def m(self):  
        print("m of B called")  
  
class C(A):  
    def m(self):  
        print("m of C called")  
  
class D(C,B):  
    pass  
  
d = D()  
d.m()
```

The Diamond Problem

```
class A:  
    def m(self):  
        print("m of A called")  
  
class B(A):  
    def m(self):  
        print("m of B called")  
  
class C(A):  
    def m(self):  
        print("m of C called")  
  
class D(C,B):  
    pass  
  
d = D()  
d.m()
```

m of C called

The Diamond Problem

```
class A:  
    def m(self):  
        print("m of A called")  
  
class B(A):  
    def m(self):  
        print("m of B called")  
  
class C(A):  
    pass  
  
class D(C,B):  
    pass  
  
d = D()  
d.m()
```

m of B called

The Diamond Problem

Now let's assume that the method m of D should execute the code of m of B, C and A as well

```
class A:  
    def m(self):  
        print("m of A called")  
  
class B(A):  
    def m(self):  
        print("m of B called")  
        super().m()  
  
class C(A):  
    def m(self):  
        print("m of C called")  
        super().m()  
  
class D(B,C):  
    def m(self):  
        print("m of D called")  
        super().m()  
  
d = D()  
d.m()
```

m of D called
m of B called
m of C called
m of A called

The Diamond Problem

The super function is often used when instances are initialized with the `__init__` method:

```
class A:  
    def __init__(self):  
        print("A.__init__")  
  
class B(A):  
    def __init__(self):  
        print("B.__init__")  
        super().__init__()  
  
class C(A):  
    def __init__(self):  
        print("C.__init__")  
        super().__init__()  
  
class D(B,C):  
    def __init__(self):  
        print("D.__init__")  
        super().__init__()
```

```
d = D()  
D.__init__  
B.__init__  
C.__init__  
A.__init__
```

The Diamond Problem

The super function is often used when instances are initialized with the `__init__` method:

```
class A:  
    def __init__(self):  
        print("A.__init__")  
  
class B(A):  
    def __init__(self):  
        print("B.__init__")  
        super().__init__()  
  
class C(A):  
    def __init__(self):  
        print("C.__init__")  
        super().__init__()  
  
class D(B,C):  
    def __init__(self):  
        print("D.__init__")  
        super().__init__()
```

c = C()

C.__init__
A.__init__

The Diamond Problem

The super function is often used when instances are initialized with the `__init__` method:

```
class A:  
    def __init__(self):  
        print("A.__init__")  
  
class B(A):  
    def __init__(self):  
        print("B.__init__")  
        super().__init__()  
  
class C(A):  
    def __init__(self):  
        print("C.__init__")  
        super().__init__()  
  
class D(B,C):  
    def __init__(self):  
        print("D.__init__")  
        super().__init__()
```

```
b = B()  
B.__init__  
A.__init__
```

The Diamond Problem

The super function is often used when instances are initialized with the `__init__` method:

```
class A:  
    def __init__(self):  
        print("A.__init__")  
  
class B(A):  
    def __init__(self):  
        print("B.__init__")  
        super().__init__()  
  
class C(A):  
    def __init__(self):  
        print("C.__init__")  
        super().__init__()  
  
class D(B,C):  
    def __init__(self):  
        print("D.__init__")  
        super().__init__()
```

a = A()

A.__init__

Next time:

Working Example of Multiple Inheritance

Lecture 21

ADT and Classes 5

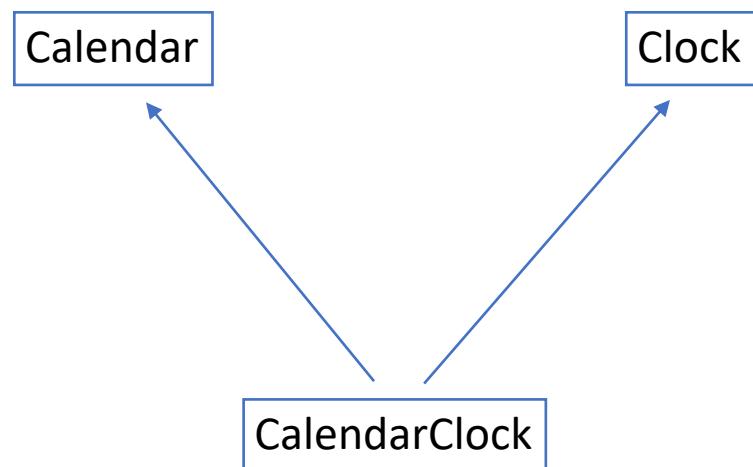
Class Hierarchies Cont.

Dr. Hacer Yalım Keleş

hkeles@ankara.edu.tr

Disclaimer: *The slide contents are mostly from https://www.python-course.eu/python3_multiple_inheritance.php*

Create class with a clock and calendar



Start with Clock class

The class Clock simulates the tick-tack of a clock.

An instance of this class contains the time, which is stored in the attributes self.hours, self.minutes and self.seconds

```
class Clock(object):

    def __init__(self, hours, minutes, seconds):
        """
        The parameters hours, minutes and seconds have to be
        integers and must satisfy the following equations:
        0 <= h < 24
        0 <= m < 60
        0 <= s < 60
        """
        self.set_Clock(hours, minutes, seconds)
```

will check the validity of the clock before setting

Start with Clock class

```
class Clock(object):
```

```
    def set_Clock(self, hours, minutes, seconds):
        """
        The parameters hours, minutes and seconds have to be
        integers and must satisfy the following equations:
        0 <= h < 24
        0 <= m < 60
        0 <= s < 60
        """

        if type(hours) == int and 0 <= hours and hours < 24:
            self._hours = hours
        else:
            raise TypeError("Hours have to be integers between 0 and 23!")
        if type(minutes) == int and 0 <= minutes and minutes < 60:
            self._minutes = minutes
        else:
            raise TypeError("Minutes have to be integers between 0 and 59!")
        if type(seconds) == int and 0 <= seconds and seconds < 60:
            self._seconds = seconds
        else:
            raise TypeError("Seconds have to be integers between 0 and 59!")
```

Start with Clock class

```
class Clock(object):
```

```
    def set_clock(self, hours, minutes, seconds):
        """
        The parameters hours, minutes and seconds have to be
        integers and must satisfy the following equations:
        0 <= h < 24
        0 <= m < 60
        0 <= s < 60
        """

        if type(hours) == int and 0 <= hours and hours < 24:
            self._hours = hours
        else:
            raise TypeError("Hours have to be integers between 0 and 23!")
        if type(minutes) == int and 0 <= minutes and minutes < 60:
            self._minutes = minutes
        else:
            raise TypeError("Minutes have to be integers between 0 and 59!")
        if type(seconds) == int and 0 <= seconds and seconds < 60:
            self._seconds = seconds
        else:
            raise TypeError("Seconds have to be integers between 0 and 59!")
```

Positional
Argument



```
    def __str__(self):
        return "{0:02d}:{1:02d}:{2:02d}".format(self._hours,
                                                self._minutes,
                                                self._seconds)
```

2 digits with
Leading zeros



Start with Clock class

```
class Clock(object):
```

```
    def tick(self):
        """
        This method lets the clock "tick", this means that the
        internal time will be advanced by one second.

        Examples:
        >>> x = Clock(12,59,59)
        >>> print(x)
        12:59:59
        >>> x.tick()
        >>> print(x)
        13:00:00
        >>> x.tick()
        >>> print(x)
        13:00:01
        """

        if self._seconds == 59:
            self._seconds = 0
            if self._minutes == 59:
                self._minutes = 0
                if self._hours == 23:
                    self._hours = 0
                else:
                    self._hours += 1
            else:
                self._minutes += 1
        else:
            self._seconds += 1
```

Start with Clock class

```
class Clock(object):
```

Class definition ...

```
if __name__ == "__main__":
    x = Clock(23,59,59)
    print(x)
    x.tick()
    print(x)
    y = str(x)
    print(type(y))
```

```
23:59:59
00:00:00
<class 'str'>
```

Calendar class

Instead of "tick" we have an "advance" method, which advances the date by one day, whenever it is called.

We have to check, if the date is the last day in a month and the number of days in the months vary.

The rules for calculating a leap year are the following:

- If a year is divisible by 400, it is a leap year.
- If a year is not divisible by 400 but by 100, it is not a leap year.
- A year number which is divisible by 4 but not by 100, it is a leap year.
- All other year numbers are common years, i.e. no leap years.

Calendar class

```
class Calendar(object):  
  
    months = (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)  
    date_style = "British"
```

```
class Calendar(object):
```

```
@staticmethod  
def leapyear(year):  
    """
```

The method leapyear returns True if the parameter year is a leap year, False otherwise

```
    """  
  
    if not year % 4 == 0:  
        return False  
    elif not year % 100 == 0:  
        return True  
    elif not year % 400 == 0:  
        return False  
    else:  
        return True
```

built-in decorator
will be explained later

```
class Calendar(object):
```

```
    def __init__(self, d, m, y):
        """
        d, m, y have to be integer values and year has to be
        a four digit year number
        """

        self.set_Calendar(d,m,y)
```

```
class Calendar(object):
```

```
    def set_Calendar(self, d, m, y):
        """
        d, m, y have to be integer values and year has to be
        a four digit year number
        """

        if type(d) == int and type(m) == int and type(y) == int:
            self.__days = d
            self.__months = m
            self.__years = y
        else:
            raise TypeError("d, m, y have to be integers!")
```

```
class Calendar(object):
```

```
    def __str__(self):
        if Calendar.date_style == "British":
            return "{0:02d}/{1:02d}/{2:4d}".format(self.__days,
                                                    self.__months,
                                                    self.__years)
        else:
            # assuming American style
            return "{0:02d}/{1:02d}/{2:4d}".format(self.__months,
                                                    self.__days,
                                                    self.__years)
```

```
class Calendar(object):
```

```
    def advance(self):
        """
        This method advances to the next date.
        """

        max_days = Calendar.months[self.__months-1]
        if self.__months == 2 and Calendar.leapyear(self.__years):
            max_days += 1
        if self.__days == max_days:
            self.__days= 1
            if self.__months == 12:
                self.__months = 1
                self.__years += 1
            else:
                self.__months += 1
        else:
            self.__days += 1
```

```
class Calendar(object):
```

```
if __name__ == "__main__":
    x = Calendar(31,12,2012)
    print(x, end=" ")
    x.advance()
    print("after applying advance: ", x)
    print("2012 was a leapyear:")
    x = Calendar(28,2,2012)
    print(x, end=" ")
    x.advance()
    print("after applying advance: ", x)
    x = Calendar(28,2,2013)
    print(x, end=" ")
    x.advance()
    print("after applying advance: ", x)
    print("1900 no leapyear: number divisible by 100 but not by 400: ")
    x = Calendar(28,2,1900)
    print(x, end=" ")
    x.advance()
    print("after applying advance: ", x)
    print("2000 was a leapyear, because number divisible by 400: ")
    x = Calendar(28,2,2000)
    print(x, end=" ")
    x.advance()
    print("after applying advance: ", x)
    print("Switching to American date style:")
    Calendar.date_style = "American"
    print("after applying advance: ", x)
```

```
31/12/2012 after applying advance: 01/01/2013
2012 was a leapyear:
28/02/2012 after applying advance: 29/02/2012
28/02/2013 after applying advance: 01/03/2013
1900 no leapyear: number divisible by 100 but not by 400:
28/02/1900 after applying advance: 01/03/1900
2000 was a leapyear, because number divisible by 400:
28/02/2000 after applying advance: 29/02/2000
Switching to American date style:
after applying advance: 02/29/2000
```

CalendarClock class

```
from clock import Clock
from calendar import Calendar

class CalendarClock(Clock, Calendar):
    """
        The class CalendarClock implements a clock with integrated
        calendar. It's a case of multiple inheritance, as it inherits
        both from Clock and Calendar
    """

    def __init__(self, day, month, year, hour, minute, second):
        Clock.__init__(self, hour, minute, second)
        Calendar.__init__(self, day, month, year)
```

```
class CalendarClock(Clock, Calendar):
```

```
    def tick(self):
        """
        advance the clock by one second
        """
        previous_hour = self._hours
        Clock.tick(self)
        if (self._hours < previous_hour):
            self.advance()
```

Alternatively:

```
super().tick()
```

```
class CalendarClock(Clock, Calendar):
```

```
def tick(self):
    """
    advance the clock by one second
    """
    previous_hour = self._hours
    Clock.tick(self)
    if (self._hours < previous_hour):
        self.advance()
```

Alternatively:

```
super().tick()
```

```
def __str__(self):
    return Calendar.__str__(self) + ", " + Clock.__str__(self)
```

```
class CalendarClock(Clock, Calendar):  
  
    def __init__(self, year, month, day, hour, minute, second):  
        self.year = year  
        self.month = month  
        self.day = day  
        self.hour = hour  
        self.minute = minute  
        self.second = second  
  
    def tick(self):  
        self.second += 1  
        if self.second == 60:  
            self.minute += 1  
            self.second = 0  
            if self.minute == 60:  
                self.hour += 1  
                self.minute = 0  
                if self.hour == 24:  
                    self.day += 1  
                    self.hour = 0  
                    if self.day == 32:  
                        self.month += 1  
                        self.day = 1  
                        if self.month == 13:  
                            self.year += 1  
                            self.month = 1  
  
    def to_string(self):  
        return "%d/%d/%d, %d:%d:%d" % (self.year, self.month, self.day, self.hour, self.minute, self.second)
```

if __name__ == "__main__":	One tick from 31/12/2013, 23:59:59 to 01/01/2014, 00:00:00
x = CalendarClock(31, 12, 2013, 23, 59, 59)	One tick from 28/02/1900, 23:59:59 to 01/03/1900, 00:00:00
print("One tick from ",x, end=" ")	One tick from 28/02/2000, 23:59:59 to 29/02/2000, 00:00:00
x.tick()	One tick from 07/02/2013, 13:55:40 to 07/02/2013, 13:55:41
print("to ", x)	
x = CalendarClock(28, 2, 1900, 23, 59, 59)	
print("One tick from ",x, end=" ")	
x.tick()	
print("to ", x)	
x = CalendarClock(28, 2, 2000, 23, 59, 59)	
print("One tick from ",x, end=" ")	
x.tick()	
print("to ", x)	
x = CalendarClock(7, 2, 2013, 13, 55, 40)	
print("One tick from ",x, end=" ")	
x.tick()	
print("to ", x)	

Next time: Decorators

Lecture 22

Function Decorators

Dr. Hacer Yalım Keleş

hkeles@ankara.edu.tr

Disclaimer: *Most of the slide contents are adapted from <https://realpython.com/primer-on-python-decorators/>*

Functions - Recap

Functions are first-class objects:

- can be passed as arguments to other functions
- can be assigned to variables similar to other objects

Functions - Recap

Functions are first-class objects:

- can be passed as arguments to other functions
- can be assigned to variables similar to other objects

```
def addone(n): → A regular function
    return n+1
```

```
def multby2(n): → A regular function
    return n*2
```

```
def compute(func,n): → A function taking another function as argument
    return func(n)
```

Functions - Recap

Functions are first-class objects:

- can be passed as arguments to other functions
- can be assigned to variables similar to other objects

```
def addone(n):
    return n+1

def multby2(n):
    return n*2

def compute(func,n):
    return func(n)
```

```
compute(addone,3)
```

```
4
```

Functions - Recap

Functions are first-class objects:

- can be passed as arguments to other functions
- can be assigned to variables similar to other objects

```
def addone(n):
    return n+1

def multby2(n):
    return n*2

def compute(func,n):
    return func(n)
```

compute(multby2,3)

6

Functions

Recap: Inner functions

```
def outer_func(n):
    print("in the outer_func() function")

    def f1(n):
        print(f"in the f1({n}) function call")

    def f2(n):
        print(f"in the f2({n}) function call")

f2(n)
f1(n)
```

Functions

Recap: Inner functions

```
def outer_func(n):
    print("in the outer_func() function")

    def f1(n):
        print(f"in the f1({n}) function call")

    def f2(n):
        print(f"in the f2({n}) function call")

    f2(n)
    f1(n)
```

```
outer_func(3)
```

```
in the outer_func() function
in the f2(3) function call
in the f1(3) function call
```

Functions

Recap: Inner functions

```
def outer_func(n):
    print("in the outer_func() function")

    def f1(n):
        print(f"in the f1({n}) function call")

    def f2(n):
        print(f"in the f2({n}) function call")

    f2(n)
    f1(n)
```

a side note:

an f-string is a literal string, prefixed with 'f', which contains expressions inside braces.
The expressions are replaced with their values.

outer_func(3)

in the outer_func() function
in the f2(3) function call
in the f1(3) function call

Functions

Recap: Inner functions

```
def outer_func(n):
    print("in the outer_func() function")

    def f1(n):
        print(f"in the f1({n}) function call")

    def f2(n):
        print(f"in the f2({n}) function call")

    if n%2==0:
        return f2
    else:
        return f1
```

Functions

Recap: Inner functions

```
def outer_func(n):
    print("in the outer_func() function")

    def f1(n):
        print(f"in the f1({n}) function call")

    def f2(n):
        print(f"in the f2({n}) function call")

    if n%2==0:
        return f2
    else:
        return f1
```

```
f = outer_func(13)
f(5)
```

```
in the outer_func() function
in the f1(5) function call
```

Functions

Recap: Inner functions

```
def outer_func(n):
    print("in the outer_func() function")

    def f1(n):
        print(f"in the f1({n}) function call")

    def f2(n):
        print(f"in the f2({n}) function call")

    if n%2==0:
        return f2
    else:
        return f1
```

```
g = outer_func(10)
g(7)
```

```
in the outer_func() function
in the f2(7) function call
```

Note that you are returning f2 without the parentheses.

Recall that this means that you are **returning a reference to the function f2**.

In contrast f2(7) with parentheses refers to the result of evaluating the function.

Functions

Recap: Inner functions

```
def outer_func(n):
    print("in the outer_func() function")

    def f1(n):
        print(f"in the f1({n}) function call")

    def f2(n):
        print(f"in the f2({n}) function call")

    if n%2==0:
        return f2
    else:
        return f1
```

```
g = outer_func(10)
g(7)
```

```
in the outer_func() function
in the f2(7) function call
```

Note that you are returning f2 without the parentheses.
Recall that this means that you are **returning a reference to the function f2**.

```
print(f)
print(g)
```

```
<function outer_func.<locals>.f1 at 0x7fd45c305ca0>
<function outer_func.<locals>.f2 at 0x7fd45c3058b0>
```

Simple Decorators

```
def my_decorator(func):
    def wrapper():
        print("before the function is called.")
        func()
        print("after the function is called.")
    return wrapper

def say_hello():
    print("hello!")

say_hello = my_decorator(say_hello)
```

Simple Decorators

```
def my_decorator(func):
    def wrapper():
        print("before the function is called.")
        func()
        print("after the function is called.")
    return wrapper

def say_hello():
    print("hello!")

say_hello = my_decorator(say_hello)
```

```
say_hello()
```

```
before the function is called.
hello!
after the function is called.
```

Simple Decorators

```
def my_decorator(func):
    def wrapper():
        print("before the function is called.")
        func()
        print("after the function is called.")
    return wrapper

def say_hello():
    print("hello!")

say_hello = my_decorator(say_hello)
```

decorating happens here!

```
say_hello()
```

```
before the function is called.
hello!
after the function is called.
```

Simple Decorators

```
def my_decorator(func):
    def wrapper():
        print("before the function is called.")
        func()
        print("after the function is called.")
    return wrapper

def say_hello():
    print("hello!")

say_hello = my_decorator(say_hello)
```

decorating happens here!

say_hello()

before the function is called.
hello!
after the function is called.

say_hello now points to the wrapper() inner function

```
print(say_hello)
```

```
<function my_decorator.<locals>.wrapper at 0x7fd45c1224c0>
```

Simple Decorators

decorators wrap a function, modifying its behavior

Simple Decorators

Second example

Write a decorator that runs a function in the day time
and do not run it at night time.

```
from datetime import datetime
def not_during_the_night(func):
    def wrapper():
        t = datetime.now().hour
        if t >= 7 and t < 22:
            func()
        else:
            pass # Hush, the neighbors are asleep
    return wrapper

def say_hello():
    print("hello!")

say_hello = not_during_the_night(say_hello)
```

when it is 2pm:

say_hello()

hello!

Simple Decorators

Second example

Write a decorator that runs a function in the day time
and do not run it at night time.

```
from datetime import datetime
def not_during_the_night(func):
    def wrapper():
        t = datetime.now().hour
        if t >= 7 and t < 22:
            func()
        else:
            pass # Hush, the neighbors are asleep
    return wrapper

def say_hello():
    print("hello!")

say_hello = not_during_the_night(say_hello)
```

when it is 23pm:

say_hello()

Nothing is printed!

Simple Decorators

Second example

Write a decorator that runs a function in the day time and do not run it at night time.

```
from datetime import datetime
def not_during_the_night(func):
    def wrapper():
        t = datetime.now().hour
        if t >= 7 and t < 22:
            func()
        else:
            pass # Hush, the neighbors are asleep
    return wrapper

def say_hello():
    print("hello!")

say_hello = not_during_the_night(say_hello)
```

The way you decorated `say_hello()` above is a little clunky.

First of all, you end up typing the name `say_hello` three times.

In addition, the decoration gets a bit hidden away below the definition of the function.

Simple Decorators

Second example

Write a decorator that runs a function in the day time and do not run it at night time.

Python allows you to use decorators in a simpler way with the @ symbol

```
def my_decorator(func):
    def wrapper():
        print("before the function is called.")
        func()
        print("after the function is called.")
    return wrapper

@my_decorator
def say_hello():
    print("hello!")
```

it performs the exact same thing with the prev. example

Syntactic sugar

Simple Decorators

Second example

Write a decorator that runs a function in the day time and do not run it at night time.

Python allows you to use decorators in a simpler way with the @ symbol

```
def my_decorator(func):
    def wrapper():
        print("before the function is called.")
        func()
        print("after the function is called.")
    return wrapper

@my_decorator
def say_hello():
    print("hello!")
```

say_hello()

```
before the function is called.
hello!
after the function is called.
```

So, @my_decorator is just an easier way of saying say_hello = my_decorator(say_hello). It's how we apply a decorator to a function.

Reuse of Decorators

decorators.py

```
def do_twice(func):
    def wrapper_do_twice():
        func()
        func()
    return wrapper_do_twice
```

Reuse of Decorators

decorators.py

```
def do_twice(func):
    def wrapper_do_twice():
        func()
        func()
    return wrapper_do_twice
```

In your file:

```
from decorators import do_twice

@do_twice
def say_hello():
    print("hello!")
```

say_hello()
hello!
hello!

Decorating functions with some arguments

decorators.py

```
def do_twice(func):
    def wrapper_do_twice():
        func()
        func()
    return wrapper_do_twice
```

No argument!

In your file:

```
from decorators import do_twice

@do_twice
def greet(name):
    print(f"Hello {name}")
```

Takes an argument!

Decorating functions with some arguments

decorators.py

```
def do_twice(func):
    def wrapper_do_twice():
        func()
        func()
    return wrapper_do_twice
```

In your file:

```
from decorators import do_twice

@do_twice
def greet(name):
    print(f"Hello {name}")
```

```
greet("John")
-----
TypeError                                         Traceback (most recent call last)
<ipython-input-51-956cdf3b8a8a> in <module>
      1 greet("John")
TypeError: wrapper_do_twice() takes 0 positional arguments but 1 was given
```

Decorating functions with some arguments

decorators.py

```
def do_twice(func):
    def wrapper_do_twice():
        func()
        func()
    return wrapper_do_twice
```

In your file:

```
from decorators import do_twice

@do_twice
def greet(name):
    print(f"Hello {name}")
```

The solution is to use `*args` and `**kwargs` in the inner wrapper function.
Then it will accept an arbitrary number of positional and keyword arguments.

```
greet("John")
-----
TypeError                                         Traceback (most recent call last)
<ipython-input-51-956cdf3b8a8a> in <module>
      1 greet("John")

TypeError: wrapper_do_twice() takes 0 positional arguments but 1 was given
```

Decorating functions with some arguments

decorators.py

Rewrite do_twice decorator function:

```
def do_twice(func):
    def wrapper_do_twice(*args, **kwargs):
        func(*args, **kwargs)
        func(*args, **kwargs)
    return wrapper_do_twice
```

In your file:

```
from decorators import do_twice

@do_twice
def greet(name):
    print(f"Hello {name}")
```

```
greet("John")
Hello John
Hello John
```

Problem solved!

Returning values

decorators.py

Rewrite do_twice decorator function:

```
def do_twice(func):
    def wrapper_do_twice(*args, **kwargs):
        func(*args, **kwargs)
        func(*args, **kwargs)
    return wrapper_do_twice
```

In your file:

```
@do_twice
def return_greeting(name):
    print("Creating greeting")
    return f"Hi {name}"
```

```
hi_adam = return_greeting("Adam")
```

```
Creating greeting
Creating greeting
```

```
print(hi_adam)
```

```
None
```

Returning values

decorators.py

Rewrite do_twice decorator function:

```
def do_twice(func):
    def wrapper_do_twice(*args, **kwargs):
        func(*args, **kwargs)
        return func(*args, **kwargs)
    return wrapper_do_twice
```

→ Returns the value returned from func

In your file:

```
@do_twice
def return_greeting(name):
    print("Creating greeting")
    return f"Hi {name}"
```

```
hi_adam = return_greeting("Adam")
```

```
Creating greeting
Creating greeting
```

```
print(hi_adam)
```

```
Hi Adam
```

Example

```
def debug(func):
    """Print the function signature and return value"""

    def wrapper_debug(*args, **kwargs):
        """This is docstring for wrapper_debug function"""
        args_repr = [repr(a) for a in args]                      # 1
        kwargs_repr = [f"{k}={v!r}" for k, v in kwargs.items()]   # 2
        signature = ", ".join(args_repr + kwargs_repr)           # 3
        print(f"Calling {func.__name__}({signature})")
        value = func(*args, **kwargs)
        print(f"{func.__name__!r} returned {value!r}")            # 4
        return value
    return wrapper_debug
```

#1: Create a list of the positional arguments. Use repr() to get a nice string representing each argument.

Example

```
def debug(func):
    """Print the function signature and return value"""

    def wrapper_debug(*args, **kwargs):
        """This is docstring for wrapper_debug function"""
        args_repr = [repr(a) for a in args]                      # 1
        kwargs_repr = [f"{k}={v!r}" for k, v in kwargs.items()]  # 2
        signature = ", ".join(args_repr + kwargs_repr)          # 3
        print(f"Calling {func.__name__}({signature})")
        value = func(*args, **kwargs)
        print(f"{func.__name__} returned {value!r}")             # 4
        return value
    return wrapper_debug
```

#2: Create a list of the keyword arguments. The f-string formats each argument as key=value where the !r specifier means that repr() is used to represent the value.

Example

```
def debug(func):
    """Print the function signature and return value"""

    def wrapper_debug(*args, **kwargs):
        """This is docstring for wrapper_debug function"""
        args_repr = [repr(a) for a in args]                      # 1
        kwargs_repr = [f"{k}={v!r}" for k, v in kwargs.items()]   # 2
        signature = ", ".join(args_repr + kwargs_repr)           # 3
        print(f"Calling {func.__name__}({signature})")
        value = func(*args, **kwargs)
        print(f"{func.__name__} returned {value!r}")               # 4
        return value
    return wrapper_debug
```

#3: The lists of positional and keyword arguments is joined together to one signature string with each argument separated by a comma.

Example

```
def debug(func):
    """Print the function signature and return value"""

    def wrapper_debug(*args, **kwargs):
        """This is docstring for wrapper_debug function"""
        args_repr = [repr(a) for a in args]                      # 1
        kwargs_repr = [f"{k}={v!r}" for k, v in kwargs.items()]   # 2
        signature = ", ".join(args_repr + kwargs_repr)           # 3
        print(f"Calling {func.__name__}({signature})")
        value = func(*args, **kwargs)
        print(f"{func.__name__!r} returned {value!r}")            # 4
        return value
    return wrapper_debug
```

#4: The return value is printed after the function is executed.

Example

```
def debug(func):
    """Print the function signature and return value"""

    def wrapper_debug(*args, **kwargs):
        """This is docstring for wrapper_debug function"""
        args_repr = [repr(a) for a in args]                      # 1
        kwargs_repr = [f"{k}={v!r}" for k, v in kwargs.items()]  # 2
        signature = ", ".join(args_repr + kwargs_repr)          # 3
        print(f"Calling {func.__name__}({signature})")
        value = func(*args, **kwargs)
        print(f"{func.__name__!r} returned {value!r}")           # 4
        return value

    return wrapper_debug
```

```
@debug
def greeting(name, surname=None):
    """This is docstring for greeting function"""
    if surname is None:
        return f"Hi {name}"
    else:
        return f"Hi {name} {surname}"
```

Example

```
def debug(func):
    """Print the function signature and return value"""

    def wrapper_debug(*args, **kwargs):
        """This is docstring for wrapper_debug function"""
        args_repr = [repr(a) for a in args]                      # 1
        kwargs_repr = [f'{k}={v!r}' for k, v in kwargs.items()]   # 2
        signature = ', '.join(args_repr + kwargs_repr)           # 3
        print(f"Calling {func.__name__}({signature})")
        value = func(*args, **kwargs)
        print(f"{func.__name__!r} returned {value!r}")            # 4
        return value

    return wrapper_debug
```

```
@debug
def greeting(name, surname=None):
    """This is docstring for greeting function"""
    if surname is None:
        return f"Hi {name}"
    else:
        return f"Hi {name} {surname}"
```

```
s = greeting("Annie")  
Calling greeting('Annie')  
'greeting' returned 'Hi Annie'
```

```
s = greeting("Annie", surname = "Maps")  
Calling greeting('Annie', surname='Maps')  
'greeting' returned 'Hi Annie Maps'
```

Example

```
def debug(func):
    """Print the function signature and return value"""

    def wrapper_debug(*args, **kwargs):
        """This is docstring for wrapper_debug function"""
        args_repr = [repr(a) for a in args] # 1
        kwargs_repr = [f"{k}={v!r}" for k, v in kwargs.items()] # 2
        signature = ", ".join(args_repr + kwargs_repr) # 3
        print(f"Calling {func.__name__}({signature})")
        value = func(*args, **kwargs)
        print(f"{func.__name__!r} returned {value!r}")
        return value
    return wrapper_debug
```

```
@debug
def greeting(name, surname=None):
    """This is docstring for greeting function"""
    if surname is None:
        return f"Hi {name}"
    else:
        return f"Hi {name} {surname}"
```

```
print(greeting.__name__)
```

```
wrapper_debug
```

```
print(greeting.__doc__)
```

This is docstring for wrapper_debug function

?

Example

```
import functools

def debug(func):
    """Print the function signature and return value"""
    @functools.wraps(func) ←
        def wrapper_debug(*args, **kwargs):
            """This is docstring for wrapper_debug function"""
            args_repr = [repr(a) for a in args]
            kwargs_repr = [f"{k}={v!r}" for k, v in kwargs.items()]
            signature = ", ".join(args_repr + kwargs_repr)
            print(f"Calling {func.__name__}({signature})")
            value = func(*args, **kwargs)
            print(f"{func.__name__!r} returned {value!r}")
            return value
    return wrapper_debug
```

Example

```
import functools

def debug(func):
    """Print the function signature and return value"""
    @functools.wraps(func)
    def wrapper_debug(*args, **kwargs):
        """This is docstring for wrapper_debug function"""
        args_repr = [repr(a) for a in args]
        kwargs_repr = [f"{k}={v!r}" for k, v in kwargs.items()]
        signature = ", ".join(args_repr + kwargs_repr)
        print(f"Calling {func.__name__}({signature})")
        value = func(*args, **kwargs)
        print(f"{func.__name__!r} returned {value!r}")
        return value
    return wrapper_debug
```

```
print(greeting.__name__)
```

```
greeting
```

```
print(greeting.__doc__)
```

```
This is docstring for greeting function
```

Next time: Decorating classes

Lecture 23

Class Decorators

Dr. Hacer Yalım Keleş

hkeles@ankara.edu.tr

Disclaimer: *Most of the slide contents are adapted from <https://realpython.com/primer-on-python-decorators/>*

Recap: Function decorators

decorators.py

Rewrite do_twice decorator function:

```
def do_twice(func):
    def wrapper_do_twice(*args, **kwargs):
        func(*args, **kwargs)
        return func(*args, **kwargs)
    return wrapper_do_twice
```

→ Returns the value returned from func

In your file:

```
@do_twice
def return_greeting(name):
    print("Creating greeting")
    return f"Hi {name}"
```



```
hi_adam = return_greeting("Adam")
```

```
Creating greeting
Creating greeting
```

```
print(hi_adam)
```

```
Hi Adam
```

Decorating Classes

Today

Class decorators

There are two different ways you can use decorators on classes:

- Similar to function decorators: decorate the class methods
- Decorate the whole class

Built-in class decorators

Instance, Class, and Static Methods

@classmethod
@staticmethod
@property

```
class MyClass:  
    def method(self): _____  
        return 'instance method called', self  
  
    @classmethod  
    def classmethod(cls):  
        return 'class method called', cls  
  
    @staticmethod  
    def staticmethod():  
        return 'static method called'
```

method, is a regular *instance method*.
the method takes one parameter, `self`,
which points to an instance of `MyClass`
when the method is called

Through the `self` parameter, instance
methods can freely access attributes and
other methods on the same object.

Instance methods can also access the
class itself through the `self.__class__`
attribute. This means instance methods
can also modify class state.

`self.__class__.name`
`self.__class__.module`
`self.__class__.mro()`

Built-in class decorators

Instance, Class, and Static Methods

@classmethod
@staticmethod
@property

```
class MyClass:  
    def method(self):  
        return 'instance method called', self  
  
    @classmethod  
    def classmethod(cls):  
        return 'class method called', cls  
  
    @staticmethod  
    def staticmethod():  
        return 'static method called'
```

Instead of accepting a `self` parameter, class methods take a `cls` parameter that points to the class—and not the object instance—when the method is called.

Because the class method only has access to this `cls` argument, it can't modify object instance state.

Built-in class decorators

Instance, Class, and Static Methods

@classmethod
@staticmethod
@property

```
class MyClass:  
    def method(self):  
        return 'instance method called', self  
  
    @classmethod  
    def classmethod(cls):  
        return 'class method called', cls  
  
    @staticmethod  
    def staticmethod():  
        return 'static method called'
```

This type of method takes neither a self nor a cls parameter - but of course it's free to accept an arbitrary number of other parameters.

a static method can neither modify object state nor class state

they allow you define utility functions in your class namespace.

Built-in class decorators

Instance, Class, and Static Methods

@classmethod
@staticmethod
@property

```
class MyClass:  
    def method(self):  
        return 'instance method called', self  
  
    @classmethod  
    def classmethod(cls):  
        return 'class method called', cls  
  
    @staticmethod  
    def staticmethod():  
        return 'static method called'
```

```
obj = MyClass()  
obj.method()
```

```
('instance method called', <__main__.MyClass at 0x7fd45c31e7c0>)
```

Built-in class decorators

Instance, Class, and Static Methods

@classmethod
@staticmethod
@property

```
class MyClass:  
    def method(self):  
        return 'instance method called', self  
  
    @classmethod  
    def classmethod(cls):  
        return 'class method called', cls  
  
    @staticmethod  
    def staticmethod():  
        return 'static method called'
```

```
obj = MyClass()  
obj.method()
```

```
('instance method called', <__main__.MyClass at 0x7fd45c31e7c0>)
```

```
MyClass.method(obj)
```

```
('instance method called', <__main__.MyClass at 0x7fd45c31e7c0>)
```

Built-in class decorators

Instance, Class, and Static Methods

@classmethod
@staticmethod
@property

```
class MyClass:  
    def method(self):  
        return 'instance method called', self  
  
    @classmethod  
    def classmethod(cls):  
        return 'class method called', cls  
  
    @staticmethod  
    def staticmethod():  
        return 'static method called'
```

obj.classmethod()

```
('class method called', __main__.MyClass)
```

MyClass.classmethod()

```
('class method called', __main__.MyClass)
```

You can call the class methods using object instances.

You do not need to create object instance for class methods

dot convention passes cls parameter automatically.

Built-in class decorators

Instance, Class, and Static Methods

@classmethod
@staticmethod
@property

```
class MyClass:  
    def method(self):  
        return 'instance method called', self  
  
    @classmethod  
    def classmethod(cls):  
        return 'class method called', cls  
  
    @staticmethod  
    def staticmethod():  
        return 'static method called'
```

obj.staticmethod()

'static method called'

MyClass.staticmethod()

'static method called'

self and cls parameters are ignored.

Built-in class decorators

Instance, Class, and Static Methods

@classmethod
@staticmethod
@property

```
class MyClass:  
    def method(self):  
        return 'instance method called', self  
  
    @classmethod  
    def classmethod(cls):  
        return 'class method called', cls  
  
    @staticmethod  
    def staticmethod():  
        return 'static method called'
```

What happens when we call these methods without creating object instances?

```
>>> MyClass.classmethod()  
'class method called', <class MyClass at 0x101a2f4c8>  
  
>>> MyClass.staticmethod()  
'static method called'  
  
>>> MyClass.method()  
TypeError: unbound method method() must  
be called with MyClass instance as first  
argument (got nothing instead)
```

Built-in class decorators

Instance, Class, and Static Methods

@classmethod
@staticmethod
@property

Example:

```
class Pizza:  
    def __init__(self, ingredients):  
        self.ingredients = ingredients  
  
    def __repr__(self):  
        return f'Pizza({self.ingredients})'
```

```
Pizza(['cheese', 'tomatoes'])
```

```
Pizza(['cheese', 'tomatoes'])
```

Built-in class decorators

Instance, Class, and Static Methods

@classmethod
@staticmethod
@property

Example:

```
class Pizza:  
    def __init__(self, ingredients):  
        self.ingredients = ingredients  
  
    def __repr__(self):  
        return f'Pizza({self.ingredients})'
```

When we want to create different kind of pizza..

```
Pizza(['mozzarella', 'tomatoes'])  
Pizza(['mozzarella', 'mushroom', 'pepper'])  
Pizza(['cheese', 'olive', 'onion'])
```

it's too crowded to define the content consistently each time!

Built-in class decorators

Instance, Class, and Static Methods

@classmethod
@staticmethod
@property

```
class Pizza:  
    def __init__(self, ingredients):  
        self.ingredients = ingredients  
  
    def __repr__(self):  
        return f'Pizza({self.ingredients})'  
  
    @classmethod  
    def margherita(cls):  
        return cls(['mozzarella', 'tomatoes'])  
  
    @classmethod  
    def mushroom(cls):  
        return cls(['mushroom', 'cheese', 'pepper'])  
  
    @classmethod  
    def mediterranean(cls):  
        return cls(['olive', 'cheese', 'tomatoe', 'onion'])
```

Define pizza factories using @classmethod!

p1 = Pizza.margherita()
p2 = Pizza.margherita()
p3 = Pizza.mushroom()

Built-in class decorators

Instance, Class, and Static Methods

@classmethod
@staticmethod
@property

```
class Pizza:  
    def __init__(self, ingredients):  
        self.ingredients = ingredients  
  
    def __repr__(self):  
        return f'Pizza({self.ingredients})'  
  
    @classmethod  
    def margherita(cls):  
        return cls(['mozzarella', 'tomatoes'])  
  
    @classmethod  
    def mushroom(cls):  
        return cls(['mushroom', 'cheese', 'pepper'])  
  
    @classmethod  
    def mediterranean(cls):  
        return cls(['olive', 'cheese', 'tomatoe', 'onion'])
```

```
p1 = Pizza.margherita()  
p2 = Pizza.margherita()  
p3 = Pizza.mushroom()
```

Built-in class decorators

Instance, Class, and Static Methods

@classmethod
@staticmethod
@property

```
class Pizza:  
    def __init__(self, ingredients):  
        self.ingredients = ingredients  
  
    def __repr__(self):  
        return f'Pizza({self.ingredients})'  
  
    @classmethod  
    def margherita(cls):  
        return cls(['mozzarella', 'tomatoes'])  
  
    @classmethod  
    def mushroom(cls):  
        return cls(['mushroom', 'cheese', 'pepper'])  
  
    @classmethod  
    def mediterranean(cls):  
        return cls(['oilve', 'cheese', 'tomatoe', 'onion'])
```

Pizza.mushroom()

Pizza(['mushroom', 'cheese', 'pepper'])

Pizza.mediterranean()

Pizza(['oilve', 'cheese', 'tomatoe', 'onion'])

They all use the same `__init__` function internally
and simply provide a shortcut for remembering all of the various ingredients.

Python only allows one `__init__` method per class.
Using class methods it's possible to add as many alternative constructors as necessary.

Built-in class decorators

Instance, Class, and Static Methods

@classmethod
@staticmethod
@property

Example @staticmethod:

```
import math

class Pizza:
    def __init__(self, radius, ingredients):
        self.radius = radius
        self.ingredients = ingredients

    def __repr__(self):
        return (f'Pizza({self.radius}!r), '
               f'{self.ingredients}!r))')

    def area(self):
        return self.circle_area(self.radius)

    @staticmethod
    def circle_area(r):
        return r ** 2 * math.pi
```

Built-in class decorators

Instance, Class, and Static Methods

@classmethod
@staticmethod
@property

Example @staticmethod:

```
import math

class Pizza:
    def __init__(self, radius, ingredients):
        self.radius = radius
        self.ingredients = ingredients

    def __repr__(self):
        return (f'Pizza({self.radius!r}, '
               f'{self.ingredients!r})')

    def area(self):
        return self.circle_area(self.radius)

    @staticmethod
    def circle_area(r):
        return r ** 2 * math.pi
```

```
>>> p = Pizza(4, ['mozzarella', 'tomatoes'])
>>> p
Pizza(4, ['mozzarella', 'tomatoes'])
>>> p.area()
50.26548245743669
>>> Pizza.circle_area(4)
50.26548245743669
```

Flagging a method as a static method is not just a hint that a method won't modify class or instance state — this restriction is also enforced by the Python runtime.

Built-in class decorators

@property

```
class Circle:
    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):
        print("getting _radius")
        return self._radius

    @radius.setter
    def radius(self, value):
        print("setting _radius")
        if value >= 0:
            self._radius = value
        else:
            raise ValueError("Radius must be positive")
```

Our private variable `_radius` keeps the state of the property `radius`

All the value assignments to `self.radius`,
Calls `@radius.setter` function

Built-in class decorators

@property

```
class Circle:
    def __init__(self, radius):
        self.radius = radius

    @property
    def radius(self):
        print("getting _radius")
        return self._radius

    @_radius.setter
    def radius(self, value):
        print("setting _radius")
        if value >= 0:
            self._radius = value
        else:
            raise ValueError("Radius must be positive")
```

c = Circle(5)

setting _radius

print(c.radius)

getting _radius
5

Built-in class decorators

@property

```
class Circle:
    def __init__(self, radius):
        self.radius = radius

    @property
    def radius(self):
        print("getting _radius")
        return self._radius

    @_radius.setter
    def radius(self, value):
        print("setting _radius")
        if value >= 0:
            self._radius = value
        else:
            raise ValueError("Radius must be positive")
```

```
p = Circle(-5)
setting _radius
-----
ValueError                                                 Traceback (most recent call last)
<ipython-input-50-f46c92fea996> in <module>
----> 1 p = Circle(-5)

<ipython-input-47-30be60cee740> in __init__(self, radius)
      1 class Circle:
      2     def __init__(self, radius):
----> 3         self.radius = radius
      4
      5     @property

<ipython-input-47-30be60cee740> in radius(self, value)
      14         self._radius = value
      15     else:
----> 16         raise ValueError("Radius must be positive")

ValueError: Radius must be positive
```

Built-in class decorators

@property

```
class Circle:
    def __init__(self, radius):
        self.radius = radius

    @property
    def radius(self):
        print("getting _radius")
        return self._radius

    @_radius.setter
    def radius(self, value):
        print("setting _radius")
        if value >= 0:
            self._radius = value
        else:
            raise ValueError("Radius must be positive")

p = Circle(-5)
```

```
setting _radius
-----
ValueError                                     Traceback (most recent call last)
<ipython-input-50-f46c92fea996> in <module>
----> 1 p = Circle(-5)

<ipython-input-47-30be60cee740> in __init__(self, radius)
      1 class Circle:
      2     def __init__(self, radius):
----> 3         self.radius = radius
      4
      5     @property

<ipython-input-47-30be60cee740> in radius(self, value)
      14         self._radius = value
      15     else:
----> 16         raise ValueError("Radius must be positive")

ValueError: Radius must be positive
```

. radius is a mutable property: it can be set to a different value. However, by defining a setter method, we can do some error testing to make sure it's not set to a nonsensical negative number. Properties are accessed as attributes without parentheses.

Built-in class decorators

@property

```
class Circle:
    def __init__(self, radius):
        self.radius = radius

    @property
    def radius(self):
        return self._radius

    @radius.setter
    def radius(self, value):
        if value >= 0:
            self._radius = value
        else:
            raise ValueError("Radius must be positive")

    @property
    def area(self):
        """Calculate area inside circle"""
        return self.pi() * self.radius**2

    @staticmethod
    def pi():
        """Value of π, could use math.pi instead though"""
        return 3.1415926535
```

Even though it is defined as a method,
it can be retrieved as an attribute without parentheses.

```
c = Circle(5)
print(c.area)
78.5398163375
```

Built-in class decorators

@property

```
class Circle:
    def __init__(self, radius):
        self.radius = radius

    @property
    def radius(self):
        return self._radius

    @radius.setter
    def radius(self, value):
        if value >= 0:
            self._radius = value
        else:
            raise ValueError("Radius must be positive")

    @property
    def area(self):
        """Calculate area inside circle"""
        return self.pi() * self.radius**2

    @staticmethod
    def pi():
        """Value of π, could use math.pi instead though"""
        return 3.1415926535
```

.area is an immutable property: properties without .setter() methods can't be changed.

```
c.area = 75
```

```
-----
AttributeError                                     Traceback (most recent call last)
<ipython-input-66-e825ceb6c1cd> in <module>
      1 c.area = 75
-----
AttributeError: can't set attribute
```

Summary

- Instance methods need a class instance and can access the instance through `self`.
- Class methods don't need a class instance. They can't access the instance (`self`) but they have access to the class itself via `cls`.
- Static methods don't have access to `cls` or `self`. They work like regular functions but belong to the class's namespace.
- Static and class methods communicate and (to a certain degree) enforce developer intent about class design. This can have maintenance benefits.

Classes as decorators

Recall that the decorator syntax `@my_decorator` is just an easier way of saying

```
func = my_decorator(func)
```

Therefore, if `my_decorator` is a class, it needs to take `func` as an argument in its
`.__init__()` method.

Furthermore, the class instance needs to be `callable` so that it can stand in for the decorated function.

Classes as decorators

For a class instance to be callable, you implement the special `__call__()` method:

```
class Counter:  
    def __init__(self, start=0):  
        self.count = start  
  
    def __call__(self):  
        self.count += 1  
        print(f"Current count is {self.count}")
```

The `__call__()` method is executed each time you try to call an instance of the class:

Classes as decorators

The `__call__()` method is executed each time you try to call an instance of the class:

```
class Counter:  
    def __init__(self, start=0):  
        self.count = start  
  
    def __call__(self):  
        self.count += 1  
        print(f"Current count is {self.count}")
```

```
>>> counter = Counter()  
>>> counter()  
Current count is 1  
  
>>> counter()  
Current count is 2  
  
>>> counter.count  
2
```

Classes as decorators

a decorator class needs to implement `__init__()` and `__call__()`:

```
import functools

class CountCalls:
    def __init__(self, func):
        functools.update_wrapper(self, func)
        self.func = func → __init__() method stores a reference to the function
        self.num_calls = 0

    def __call__(self, *args, **kwargs):
        self.num_calls += 1
        print(f"Call {self.num_calls} of {self.func.__name__!r}")
        return self.func(*args, **kwargs)

@CountCalls
def say_whee():
    print("Whee!")
```

→ `__call__()` method will be called instead of the decorated function

Classes as decorators

a decorator class needs to implement `__init__()` and `__call__()`:

```
import functools

class CountCalls:
    def __init__(self, func):
        functools.update_wrapper(self, func)
        self.func = func
        self.num_calls = 0

    def __call__(self, *args, **kwargs):
        self.num_calls += 1
        print(f"Call {self.num_calls} of {self.func.__name__}!")
        return self.func(*args, **kwargs)

@CountCalls
def say_whee():
    print("Whee!")
```

```
>>> say_whee()
Call 1 of 'say_whee'
Whee!

>>> say_whee()
Call 2 of 'say_whee'
Whee!

>>> say_whee.num_calls
2
```

An example for decorating a class: Creating a singleton

A singleton is a class with only one instance.

There are several singletons in Python that you use frequently, including:

`None`, `True`, and `False`

It is the fact that `None` is a singleton that allows you to compare for `None` using the `is` keyword.

There is only a single `None` instance.

An example for decorating a class: Creating a singleton

The following `@singleton` decorator turns a class into a singleton by storing the first instance of the class as an attribute. Later attempts at creating an instance simply return the stored instance:

```
import functools

def singleton(cls):
    """Make a class a Singleton class (only one instance)"""
    @functools.wraps(cls)
    def wrapper_singleton(*args, **kwargs):
        if not wrapper_singleton.instance:
            wrapper_singleton.instance = cls(*args, **kwargs)
        return wrapper_singleton.instance
    wrapper_singleton.instance = None
    return wrapper_singleton
```

```
@singleton
class TheOne:
    pass
```

instead of func, we pass cls

An example for decorating a class: Creating a singleton

The following `@singleton` decorator turns a class into a singleton by storing the first instance of the class as an attribute. Later attempts at creating an instance simply return the stored instance:

```
import functools

def singleton(cls):
    """Make a class a Singleton class (only one instance)"""
    @functools.wraps(cls)
    def wrapper_singleton(*args, **kwargs):
        if not wrapper_singleton.instance:
            wrapper_singleton.instance = cls(*args, **kwargs)
        return wrapper_singleton.instance
    wrapper_singleton.instance = None
    return wrapper_singleton
```

```
@singleton
class TheOne:
    pass
```

```
>>> first_one = TheOne()
>>> another_one = TheOne()

>>> id(first_one)
140094218762280

>>> id(another_one)
140094218762280

>>> first_one is another_one
True
```

Lecture 24

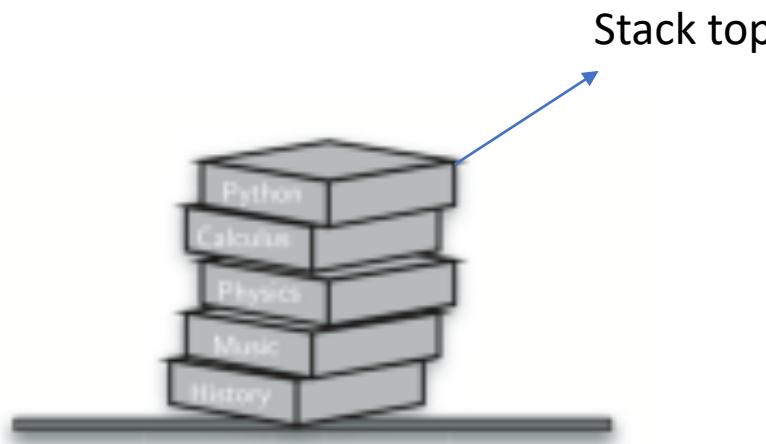
ADT - Stacks

Dr. Hacer Yalım Keleş

hkeles@ankara.edu.tr

Disclaimer: *Most of the slide contents are adapted from the Book: Problem Solving with Algorithms and Data Structures using Python*

What is a Stack?



A stack of books



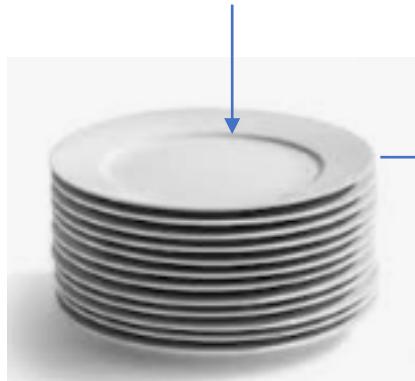
A stack of plates

What is a Stack?



A stack of books

Put new item to the top



A stack of plates

What is a Stack?



A stack of books

Remove one item from the top



A stack of plates

Methods of Stack ADT



Stack(): creates a new stack that is empty. It needs no parameters and returns an empty stack.

push(item): adds a new item to the top of the stack. It needs the item and returns nothing.

pop(): removes the top item from the stack. It needs no parameters and returns the item. The stack is modified.

peek(): returns the top item from the stack but does not remove it. It needs no parameters. The stack is not modified.

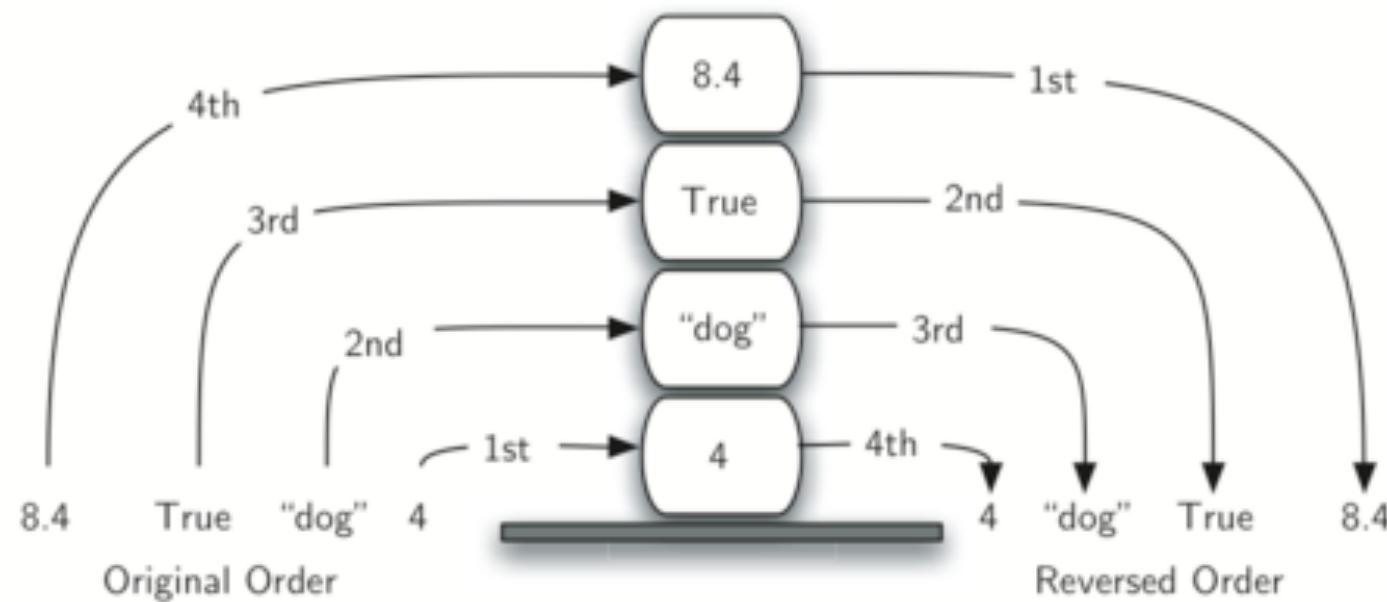
isEmpty(): tests to see whether the stack is empty. It needs no parameters and returns a boolean value.

size(): returns the number of items on the stack. It needs no parameters and returns an integer.

Stack Class

```
class Stack:  
    def __init__(self):  
        self.data = []  
  
    def isEmpty(self):  
        return self.data == []  
  
    def push(self, item):  
        self.data.append(item)  
  
    def pop(self):  
        return self.data.pop()  
  
    def peek(self):  
        return self.data[len(self.data)-1]  
  
    def size(self):  
        return len(self.data)
```

Reverse Ordering with Stacks



Use Stack Class

```
s=Stack()  
  
print(s.isEmpty())  
s.push(4)  
s.push('dog')  
print(s.peek())  
s.push(True)  
print(s.size())  
print(s.isEmpty())  
s.push(8.4)  
print(s.pop())  
print(s.pop())  
print(s.size())
```



Use Stack Class

```
s=Stack()  
  
print(s.isEmpty())  
s.push(4)  
s.push('dog')  
print(s.peek())  
s.push(True)  
print(s.size())  
print(s.isEmpty())  
s.push(8.4)  
print(s.pop())  
print(s.pop())  
print(s.size())
```



Output:

```
True  
dog  
3  
False  
8.4  
True  
2
```

Use Case: Balanced Parentheses

```
(2+3)*(5/(7-9))
```

Balanced parentheses means that each opening symbol has a corresponding closing symbol and the pairs of parentheses are properly nested

balanced

```
((()())())
```

```
((((())
```

```
((()((())())
```

unbalanced

```
(((((())
```

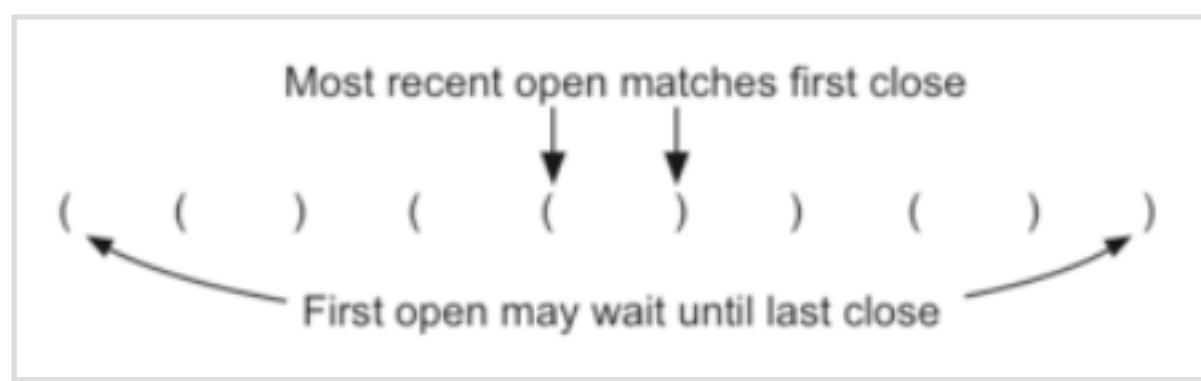
```
()))
```

```
((())()
```

The ability to differentiate between parentheses that are correctly balanced and those that are unbalanced is an important part of recognizing many programming language structures.

Use Case: Balanced Parentheses

Balanced parentheses means that each opening symbol has a corresponding closing symbol and the pairs of parentheses are properly nested



```
def parChecker(symbolString):
    s = Stack()
    balanced = True
    index = 0
    while index < len(symbolString) and balanced:
        symbol = symbolString[index]
        if symbol == "(":
            s.push(symbol)
        else:
            if s.isEmpty():
                balanced = False
            else:
                s.pop()

        index = index + 1

    if balanced and s.isEmpty():
        return True
    else:
        return False
```

```
def parChecker(symbolString):
    s = Stack()
    balanced = True
    index = 0
    while index < len(symbolString) and balanced:
        symbol = symbolString[index]
        if symbol == "(":
            s.push(symbol)
        else:
            if s.isEmpty():
                balanced = False
            else:
                s.pop()
        index = index + 1

    if balanced and s.isEmpty():
        return True
    else:
        return False
```

```
print(parChecker('(((( )))))'))
print(parChecker('(( ))'))
```

```
True
False
```

```
def parCheckerGeneral(symbolString):
    s = Stack()
    balanced = True
    index = 0
    while index < len(symbolString) and balanced:
        symbol = symbolString[index]
        if symbol in "([{":
            s.push(symbol)
        else:
            if s.isEmpty():
                balanced = False
            else:
                top = s.pop()
                if not matches(top,symbol):
                    balanced = False
        index = index + 1
    if balanced and s.isEmpty():
        return True
    else:
        return False
```

MORE GENERAL PATTERN MATCHER

```
def parCheckerGeneral(symbolString):
    s = Stack()
    balanced = True
    index = 0
    while index < len(symbolString) and balanced:
        symbol = symbolString[index]
        if symbol in "([{":
            s.push(symbol)
        else:
            if s.isEmpty():
                balanced = False
            else:
                top = s.pop()
                if not matches(top,symbol):
                    balanced = False
        index = index + 1
    if balanced and s.isEmpty():
        return True
    else:
        return False
```

```
def matches(open,close):
    opens = "([{"
    closers = ")]}"
    return opens.index(open) == closers.index(close)
```

MORE GENERAL PATTERN MATCHER

```
def parCheckerGeneral(symbolString):
    s = Stack()
    balanced = True
    index = 0
    while index < len(symbolString) and balanced:
        symbol = symbolString[index]
        if symbol in "([{":
            s.push(symbol)
        else:
            if s.isEmpty():
                balanced = False
            else:
                top = s.pop()
                if not matches(top,symbol):
                    balanced = False
        index = index + 1
    if balanced and s.isEmpty():
        return True
    else:
        return False
```

```
print(parCheckerGeneral('{{{{[]}}}}'))
```

True

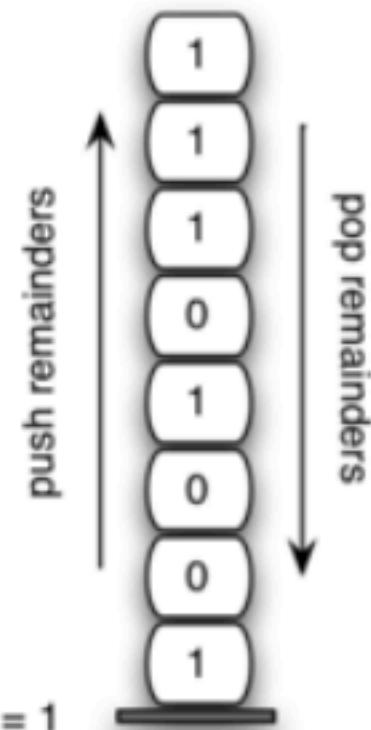
```
print(parCheckerGeneral('[{()]}'))
```

False

```
def matches(open,close):
    opens = "[({"
    closers = ")]}"
    return opens.index(open) == closers.index(close)
```

Use Case: Converting Decimal Numbers to Binary

```
233 // 2 = 116  rem = 1  
116 // 2 = 58   rem = 0  
58 // 2 = 29   rem = 0  
29 // 2 = 14   rem = 1  
14 // 2 = 7    rem = 0  
7 // 2 = 3    rem = 1  
3 // 2 = 1    rem = 1  
1 // 2 = 0    rem = 1
```



Decimal to Binary Conversion Algorithm

Use Case: Converting Decimal Numbers to Binary

```
def Decimal2Binary(decNumber):
    remstack = Stack()

    while decNumber > 0:
        rem = decNumber % 2
        remstack.push(rem)
        decNumber = decNumber // 2

    binString = ""
    while not remstack.isEmpty():
        binString = binString + str(remstack.pop())

    return binString
```

Use Case: Converting Decimal Numbers to Binary

```
def Decimal2Binary(decNumber):
    remstack = Stack()

    while decNumber > 0:
        rem = decNumber % 2
        remstack.push(rem)
        decNumber = decNumber // 2

    binString = ""
    while not remstack.isEmpty():
        binString = binString + str(remstack.pop())

    return binString
```

```
print(Decimal2Binary(4))
100

print(Decimal2Binary(233))
11101001
```

Use Case: Converting Decimal to Any Base

```
def baseConverter(decNumber,base):
    digits = "0123456789ABCDEF"

    remstack = Stack()

    while decNumber > 0:
        rem = decNumber % base
        remstack.push(rem)
        decNumber = decNumber // base

    newString = ""
    while not remstack.isEmpty():
        newString = newString + digits[remstack.pop()]

    return newString
```

Use Case: Converting Decimal to Any Base

```
def baseConverter(decNumber,base):
    digits = "0123456789ABCDEF"

    remstack = Stack()

    while decNumber > 0:
        rem = decNumber % base
        remstack.push(rem)
        decNumber = decNumber // base

    newString = ""
    while not remstack.isEmpty():
        newString = newString + digits[remstack.pop()]

    return newString
```

```
print(baseConverter(125,2))
1111101

print(baseConverter(125,8))
175

print(baseConverter(125,16))
7D
```

Lecture 25

ADT - Trees

Dr. Hacer Yalım Keleş

hkeles@ankara.edu.tr

Disclaimer: *Most of the slide contents are adapted from the Book: Problem Solving with Algorithms and Data Structures using Python*

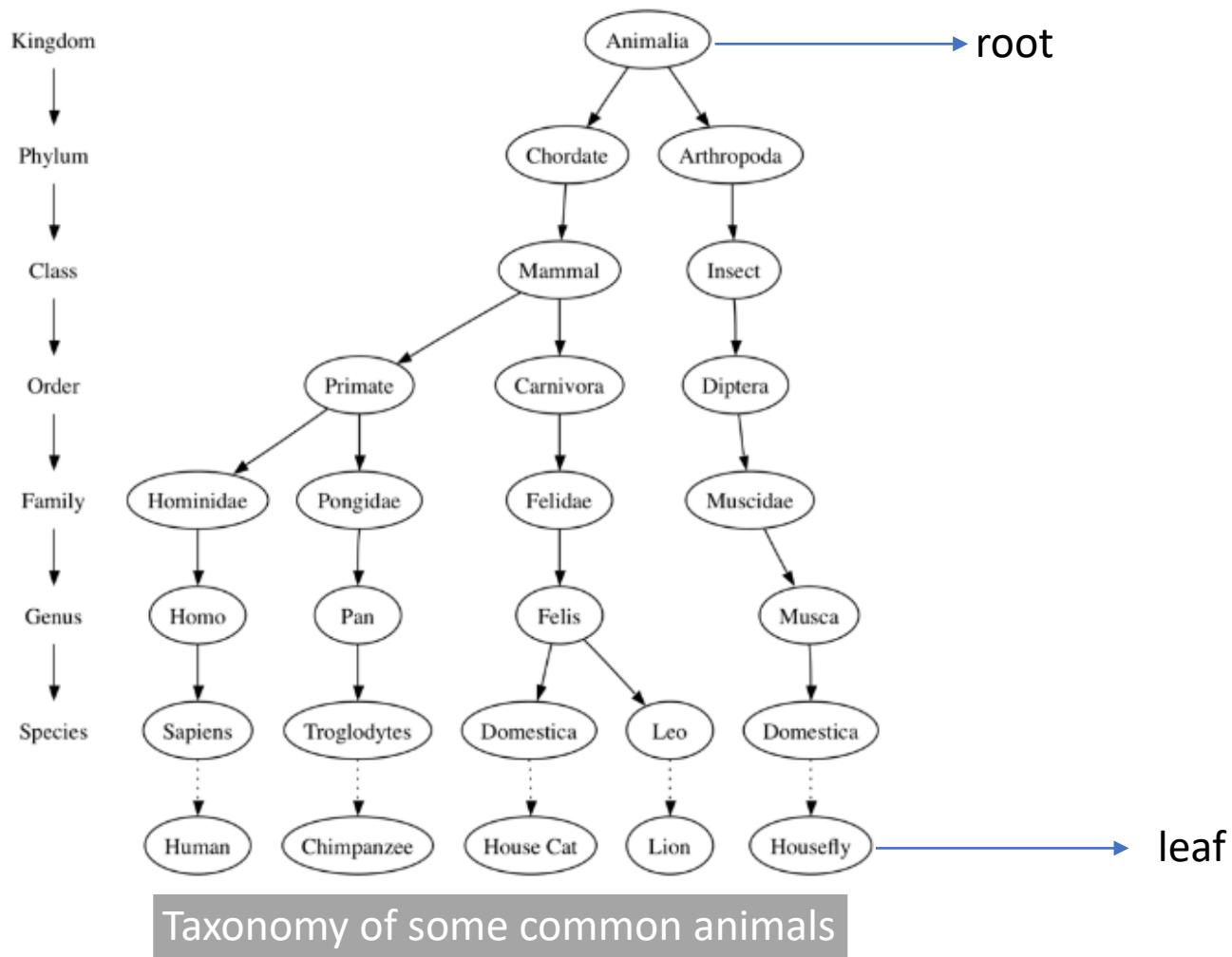
What is a Tree?

Trees are used in many areas of computer science, including:
operating systems, graphics, database systems, and computer networking.

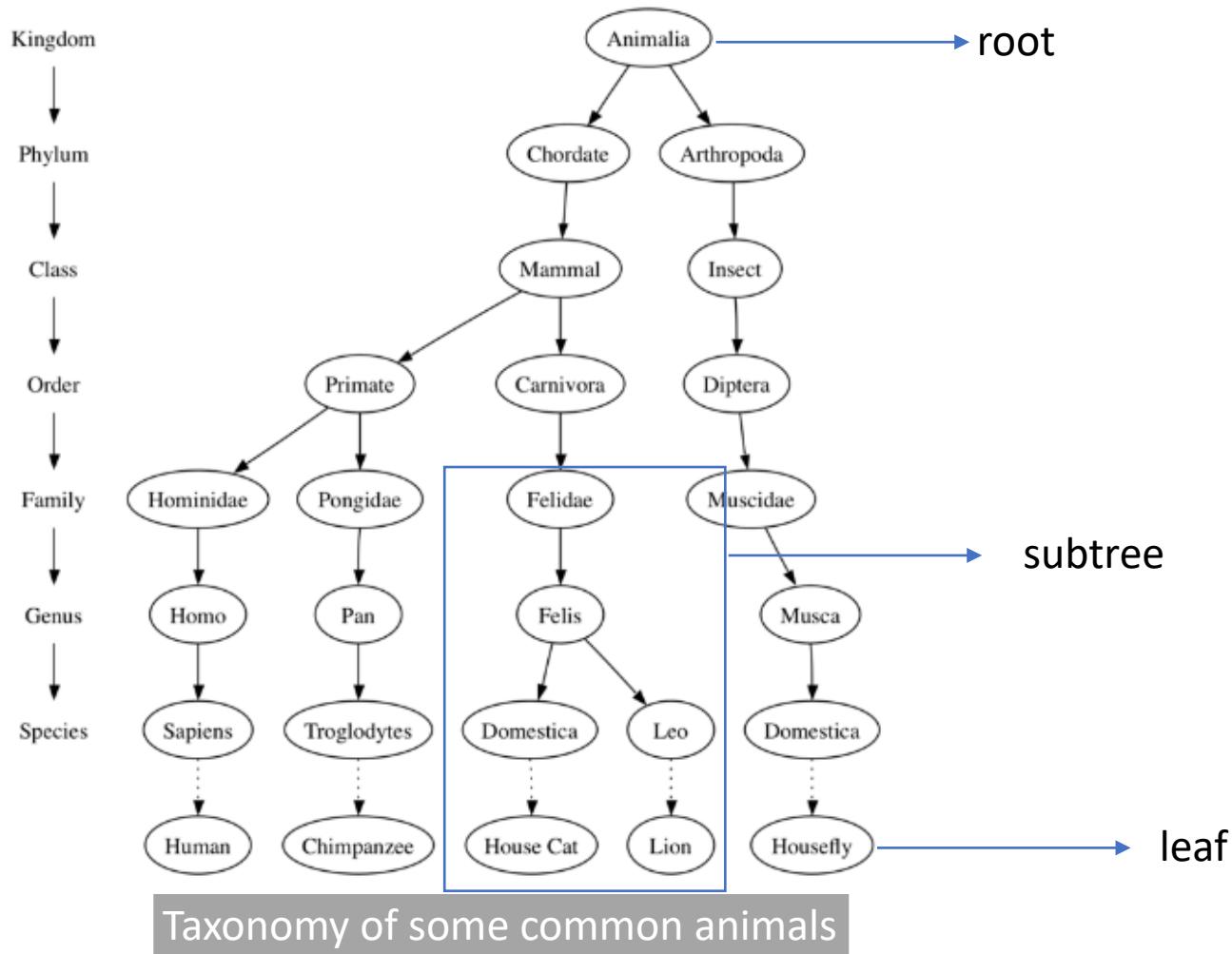
A tree data structure has a **root**, **branches**, and **leaves**

The difference between a tree in nature and a tree in computer science is that
a tree data structure has its root at the top and its leaves on the bottom.

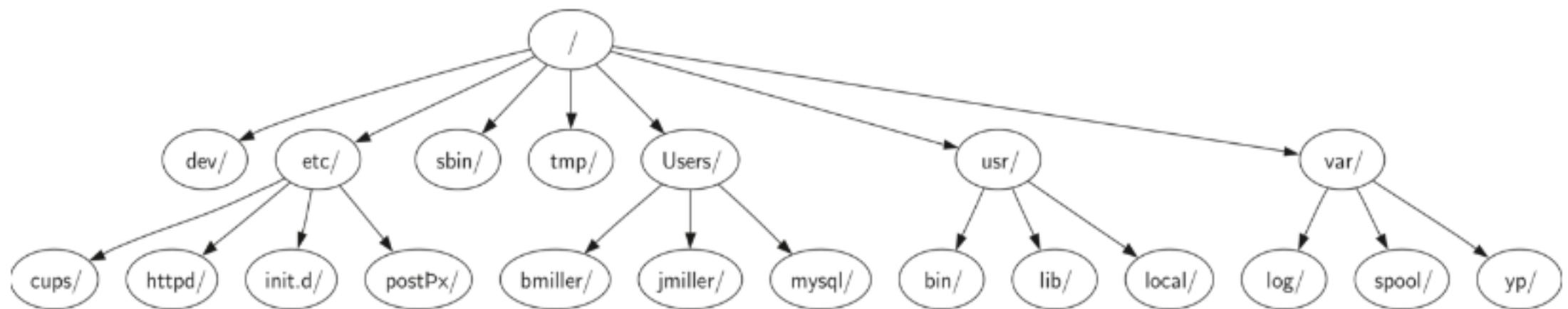
What is a Tree?



What is a Tree?



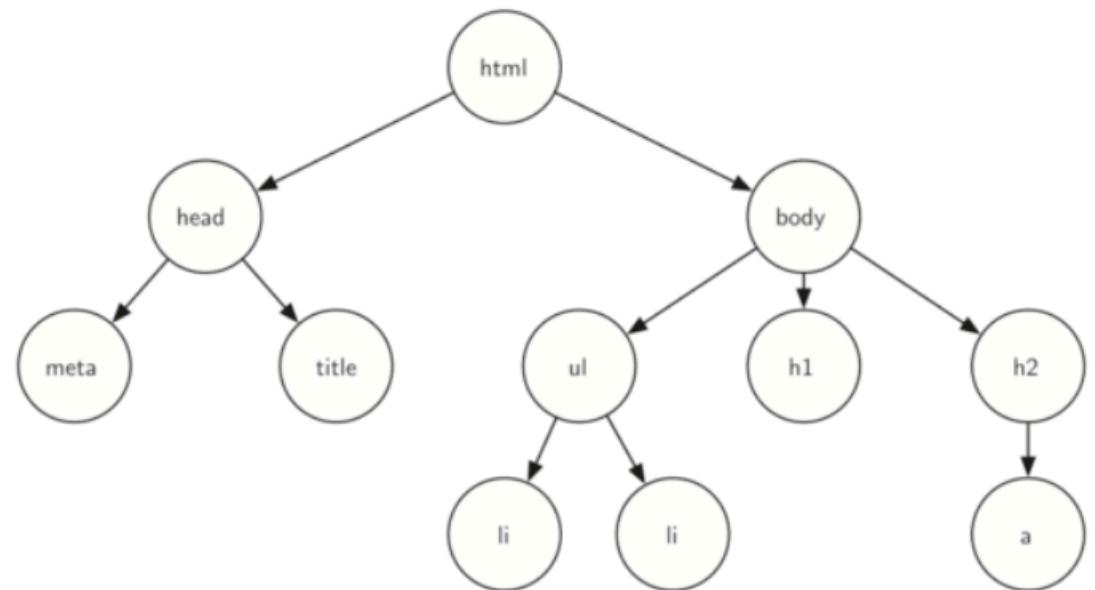
What is a Tree?



Sample Unix file system hierarchy

What is a Tree?

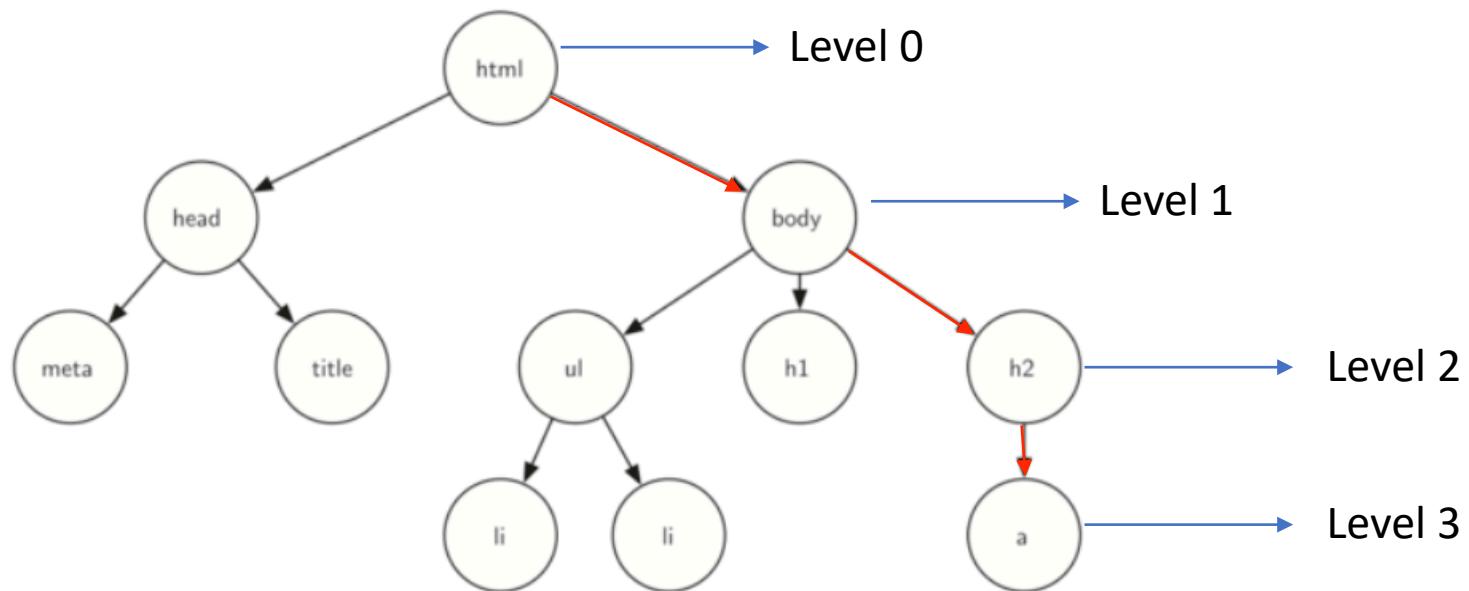
```
<html xmlns="http://www.w3.org/1999/xhtml"
      xml:lang="en" lang="en">
<head>
  <meta http-equiv="Content-Type"
        content="text/html; charset=utf-8" />
  <title>simple</title>
</head>
<body>
  <h1>A simple web page</h1>
  <ul>
    <li>List item one</li>
    <li>List item two</li>
  </ul>
  <h2><a href="http://www.cs.luther.edu">Luther CS </a></h2>
</body>
</html>
```



A sample web page created using HTML

Corresponding mark-up elements of the web page

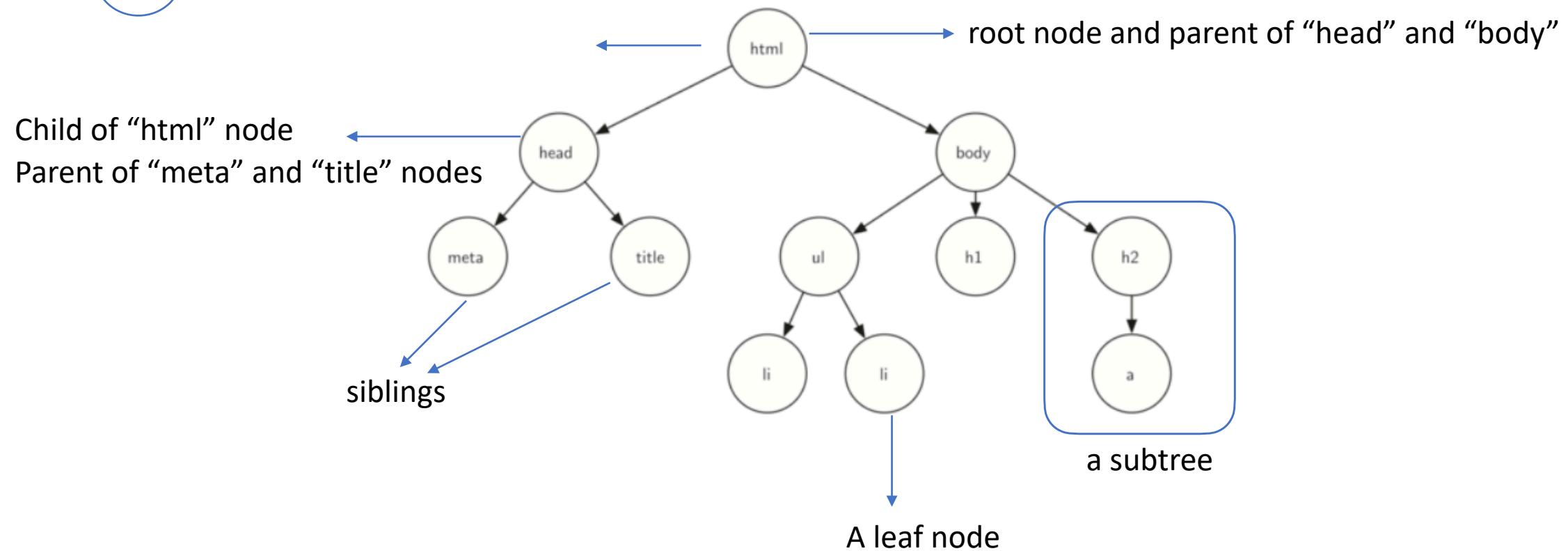
Important terms



Height of a tree: max level from root to a leaf node

Height → 3

Important terms



Python implementation (1)

List of lists tree

In a list of lists tree;

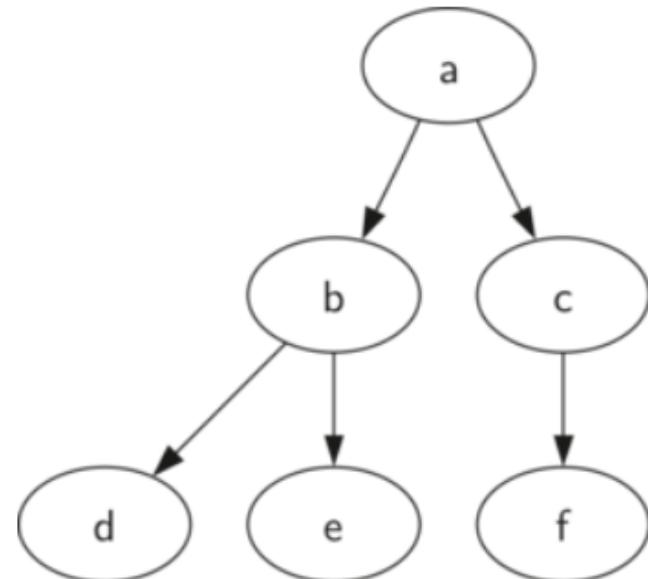
- we will store **the value of the root node** as the first element of the list.
- the second element of the list will itself be a list that represents **the left subtree**.
- the third element of the list will be another list that represents **the right subtree**.

Python implementation

List of lists tree

In a list of lists tree;

- we will store **the value of the root node** as the first element of the list.
- the second element of the list will itself be a list that represents **the left subtree**.
- the third element of the list will be another list that represents **the right subtree**.



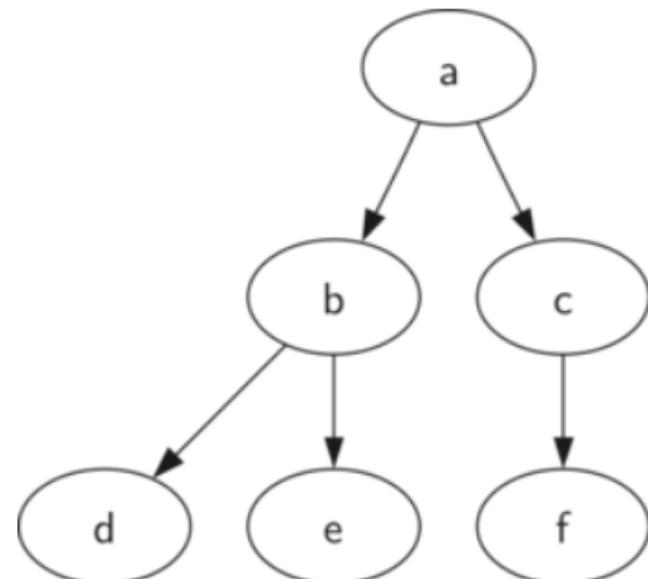
```
myTree = ['a', #root  
          ['b', #left subtree  
           ['d', [], []],  
            ['e', [], []]],  
          ['c', #right subtree  
           ['f', [], []],  
            []]]
```

Python implementation

List of lists tree

In a list of lists tree;

- we will store **the value of the root node** as the first element of the list.
- the second element of the list will itself be a list that represents **the left subtree**.
- the third element of the list will be another list that represents **the right subtree**.



```
myTree = ['a', #root  
          ['b', #left subtree  
           ['d', [], []],  
           ['e', [], []]],  
          ['c', #right subtree  
           ['f', [], []],  
           []]]
```

Recursive structure

Root → myTree[0]

Left subtree → myTree[1]

Right subtree → myTree[2]

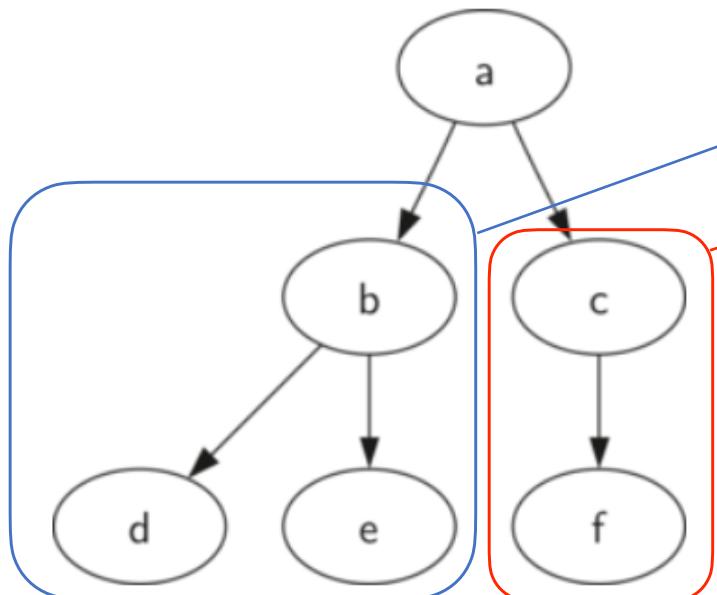
Python implementation

List of lists tree

Recursive structure

```
myTree = ['a', #root  
          ['b', #left subtree  
           ['d', [], []],  
           ['e', [], []]],  
          ['c', #right subtree  
           ['f', [], []],  
           []]  
        ]
```

```
myTree = ['a', ['b', ['d',[],[]], ['e',[],[]]], ['c', ['f',[],[]], []]]  
print(myTree)  
print('left subtree = ', myTree[1])  
print('root = ', myTree[0])  
print('right subtree = ', myTree[2])
```



```
['a', ['b', ['d', [], []], ['e', [], []]], ['c', ['f', [], []], []]]  
left subtree =  ['b', ['d', [], []], ['e', [], []]]  
root =  a  
right subtree =  ['c', ['f', [], []], []]
```

Python implementation

List of lists tree

```
def BinaryTree(r):
    return [r, [], []]
```

Python implementation

List of lists tree

```
def BinaryTree(r):
    return [r, [], []]

def insertLeft(root,newBranch):
    t = root.pop(1)
    if len(t) > 0: _____ → if there is a left node already..
        root.insert(1,[newBranch,t,[]])
    else:
        root.insert(1,[newBranch, [], []])
    return root

def insertRight(root,newBranch):
    t = root.pop(2)
    if len(t) > 0: _____ → if there is a right node already..
        root.insert(2,[newBranch,[],t])
    else:
        root.insert(2,[newBranch,[],[]])
    return root
```

Python implementation

List of lists tree

```
def BinaryTree(r):
    return [r, [], []]

def insertLeft(root,newBranch):
    t = root.pop(1)
    if len(t) > 0:
        root.insert(1,[newBranch,t,[]])
    else:
        root.insert(1,[newBranch, [], []])
    return root

def insertRight(root,newBranch):
    t = root.pop(2)
    if len(t) > 0:
        root.insert(2,[newBranch,[],t])
    else:
        root.insert(2,[newBranch,[],[]])
    return root
```

```
def getRootVal(root):
    return root[0]

def setRootVal(root,newVal):
    root[0] = newVal

def getLeftChild(root):
    return root[1]

def getRightChild(root):
    return root[2]
```

```
def BinaryTree(r):
    return [r, [], []]

def insertLeft(root,newBranch):
    t = root.pop(1)
    if len(t) > 0:
        root.insert(1,[newBranch,t,[]])
    else:
        root.insert(1,[newBranch, [], []])
    return root

def insertRight(root,newBranch):
    t = root.pop(2)
    if len(t) > 0:
        root.insert(2,[newBranch,[],t])
    else:
        root.insert(2,[newBranch,[],[]])
    return root

def getRootVal(root):
    return root[0]

def setRootVal(root,newVal):
    root[0] = newVal

def getLeftChild(root):
    return root[1]

def getRightChild(root):
    return root[2]
```

```
r = BinaryTree(3)
insertLeft(r,4)
insertLeft(r,5)
insertRight(r,6)
insertRight(r,7)
l = getLeftChild(r)
print(l)

setRootVal(l,9)
print(r)
insertLeft(l,11)
print(r)
print(getRightChild(getRightChild(r)))
```

```

def BinaryTree(r):
    return [r, [], []]

def insertLeft(root,newBranch):
    t = root.pop(1)
    if len(t) > 0:
        root.insert(1,[newBranch,t,[]])
    else:
        root.insert(1,[newBranch, [], []])
    return root

def insertRight(root,newBranch):
    t = root.pop(2)
    if len(t) > 0:
        root.insert(2,[newBranch,[],t])
    else:
        root.insert(2,[newBranch,[],[]])
    return root

def getRootVal(root):
    return root[0]

def setRootVal(root,newVal):
    root[0] = newVal

def getLeftChild(root):
    return root[1]

def getRightChild(root):
    return root[2]

```

```

r = BinaryTree(3)
insertLeft(r,4)
insertLeft(r,5)
insertRight(r,6)
insertRight(r,7)
l = getLeftChild(r)
print(l)

setRootVal(l,9)
print(r)
insertLeft(l,11)
print(r)
print(getRightChild(getRightChild(r)))

```

```

[5, [4, [], []], []]
[3, [9, [4, [], []], []], [7, [], [6, [], []]]]
[3, [9, [11, [4, [], []], []], []], [7, [], [6, [], []]]]
[6, [], []]

```

```

def BinaryTree(r):
    return [r, [], []]

def insertLeft(root,newBranch):
    t = root.pop(1)
    if len(t) > 0:
        root.insert(1,[newBranch,t,[]])
    else:
        root.insert(1,[newBranch, [], []])
    return root

def insertRight(root,newBranch):
    t = root.pop(2)
    if len(t) > 0:
        root.insert(2,[newBranch,[],t])
    else:
        root.insert(2,[newBranch,[],[]])
    return root

def getRootVal(root):
    return root[0]

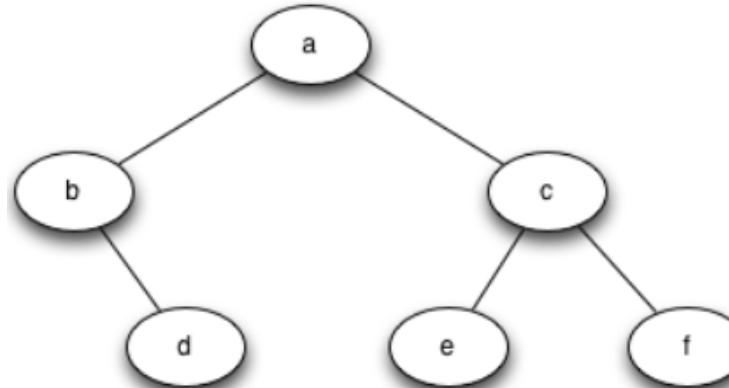
def setRootVal(root,newVal):
    root[0] = newVal

def getLeftChild(root):
    return root[1]

def getRightChild(root):
    return root[2]

```

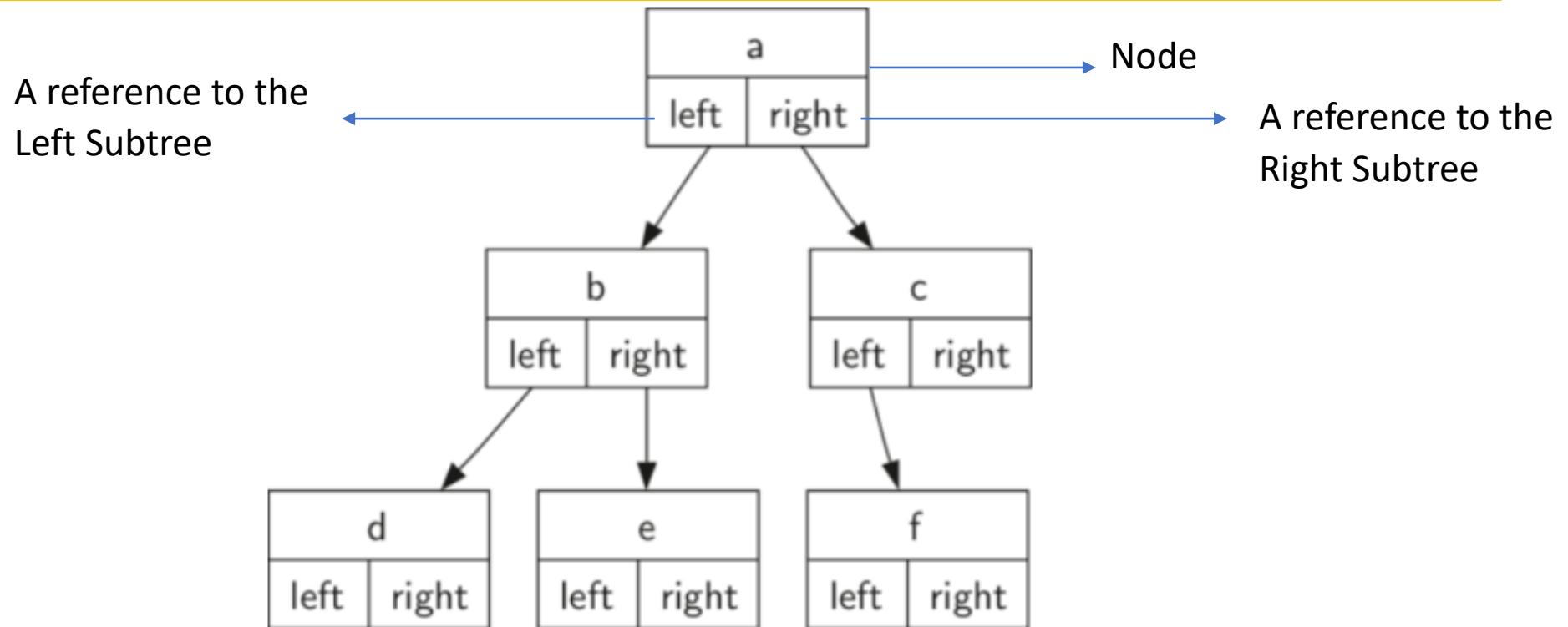
Write a function `buildTree` that returns a tree using the list of lists functions that looks like this:



Python implementation (2)

Object Oriented Approach

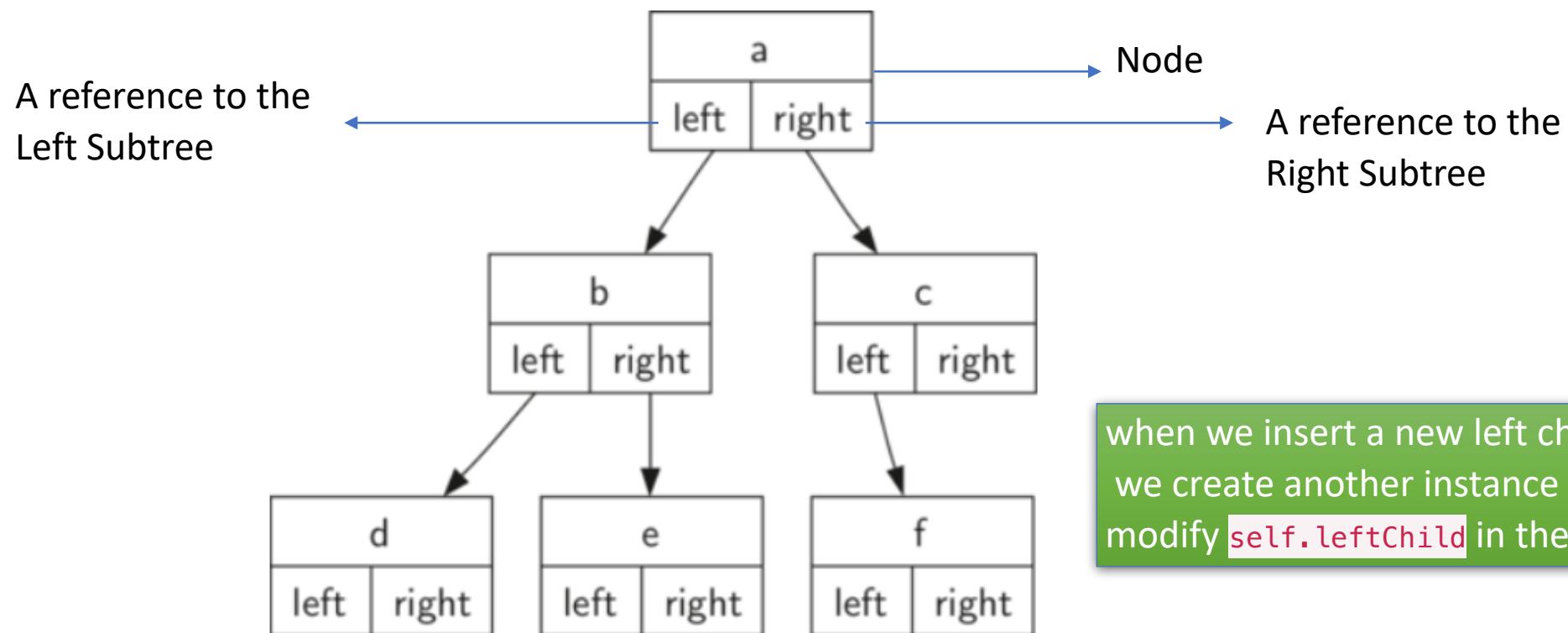
we will define a class that has attributes for the root value, as well as the left and right subtrees.



Python implementation (2)

Object Oriented Approach

we will define a class that has attributes for the root value, as well as the left and right subtrees.



when we insert a new left child into the tree,
we create another instance of `BinaryTree` and
modify `self.leftChild` in the root to reference the new tree.

Python implementation (2)

Object Oriented Approach

```
class BinaryTree:  
    def __init__(self,rootObj):  
        self.key = rootObj  
        self.leftChild = None  
        self.rightChild = None
```

the root object of a tree can be a reference to any object

when we insert a new left child into the tree we create another instance of `BinaryTree` and modify `self.leftChild` in the root to reference the new tree.

Python implementation (2)

Object Oriented Approach

```
def insertLeft(self,newNode):
    if self.leftChild == None:
        self.leftChild = BinaryTree(newNode)
    else:
        t = BinaryTree(newNode)
        t.leftChild = self.leftChild
        self.leftChild = t
```

Python implementation (2)

Object Oriented Approach

```
def insertRight(self,newNode):
    if self.rightChild == None:
        self.rightChild = BinaryTree(newNode)
    else:
        t = BinaryTree(newNode)
        t.rightChild = self.rightChild
        self.rightChild = t
```

```
class BinaryTree:  
    def __init__(self,rootObj):  
        self.key = rootObj  
        self.leftChild = None  
        self.rightChild = None  
  
    def insertLeft(self,newNode):  
        if self.leftChild == None:  
            self.leftChild = BinaryTree(newNode)  
        else:  
            t = BinaryTree(newNode)  
            t.leftChild = self.leftChild  
            self.leftChild = t  
  
    def insertRight(self,newNode):  
        if self.rightChild == None:  
            self.rightChild = BinaryTree(newNode)  
        else:  
            t = BinaryTree(newNode)  
            t.rightChild = self.rightChild  
            self.rightChild = t
```

Accessor methods:

```
def getRightChild(self):  
    return self.rightChild  
  
def getLeftChild(self):  
    return self.leftChild  
  
def setRootVal(self,obj):  
    self.key = obj  
  
def getRootVal(self):  
    return self.key
```

```
class BinaryTree:  
    def __init__(self,rootObj):  
        self.key = rootObj  
        self.leftChild = None  
        self.rightChild = None  
  
    def insertLeft(self,newNode):  
        if self.leftChild == None:  
            self.leftChild = BinaryTree(newNode)  
        else:  
            t = BinaryTree(newNode)  
            t.leftChild = self.leftChild  
            self.leftChild = t  
  
    def insertRight(self,newNode):  
        if self.rightChild == None:  
            self.rightChild = BinaryTree(newNode)  
        else:  
            t = BinaryTree(newNode)  
            t.rightChild = self.rightChild  
            self.rightChild = t
```

```
r = BinaryTree('a')  
print(r.getRootVal())  
print(r.getLeftChild())  
r.insertLeft('b')  
print(r.getLeftChild())  
print(r.getLeftChild().getRootVal())  
r.insertRight('c')  
print(r.getRightChild())  
print(r.getRightChild().getRootVal())  
r.getRightChild().setRootVal('hello')  
print(r.getRightChild().getRootVal())
```

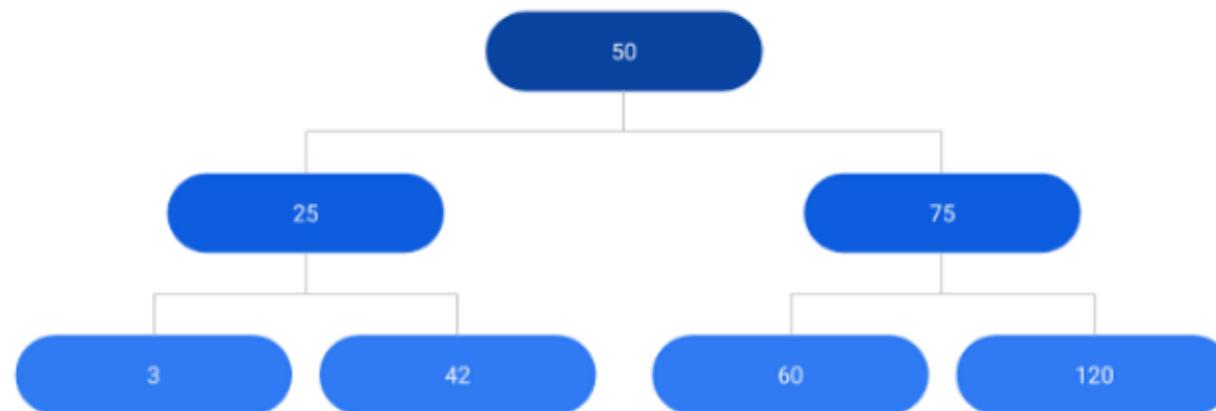
```
class BinaryTree:  
    def __init__(self,rootObj):  
        self.key = rootObj  
        self.leftChild = None  
        self.rightChild = None  
  
    def insertLeft(self,newNode):  
        if self.leftChild == None:  
            self.leftChild = BinaryTree(newNode)  
        else:  
            t = BinaryTree(newNode)  
            t.leftChild = self.leftChild  
            self.leftChild = t  
  
    def insertRight(self,newNode):  
        if self.rightChild == None:  
            self.rightChild = BinaryTree(newNode)  
        else:  
            t = BinaryTree(newNode)  
            t.rightChild = self.rightChild  
            self.rightChild = t
```

```
r = BinaryTree('a')  
print(r.getRootVal())  
print(r.getLeftChild())  
r.insertLeft('b')  
print(r.getLeftChild())  
print(r.getLeftChild().getRootVal())  
r.insertRight('c')  
print(r.getRightChild())  
print(r.getRightChild().getRootVal())  
r.getRightChild().setRootVal('hello')  
print(r.getRightChild().getRootVal())
```

```
a  
None  
<__main__.BinaryTree object>  
b  
<__main__.BinaryTree object>  
c  
hello
```

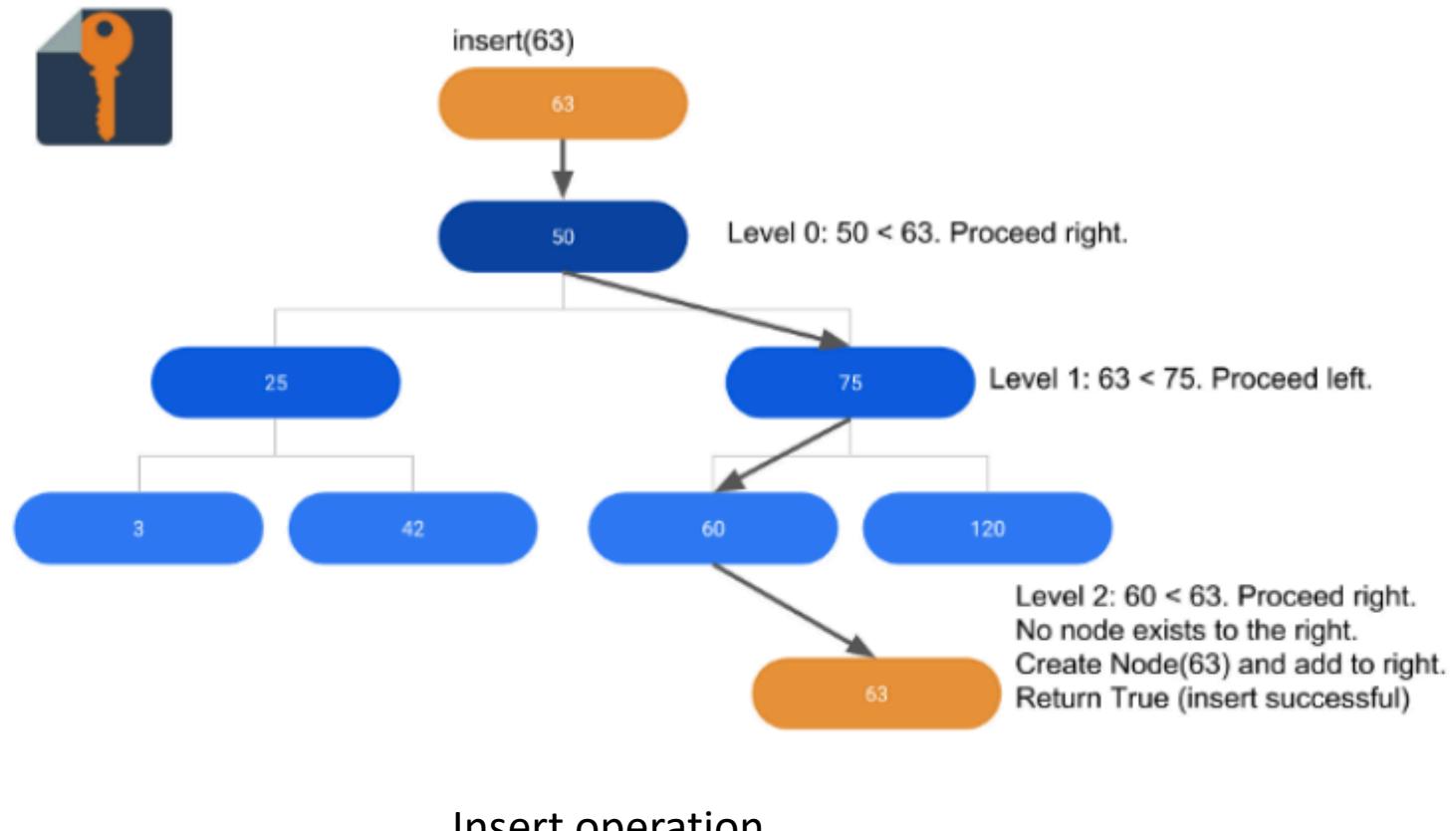
Case Study: Binary Search Trees (BST)

A binary search tree relies on the property that keys that are less than the parent are found in the left subtree, and keys that are greater than the parent are found in the right subtree.



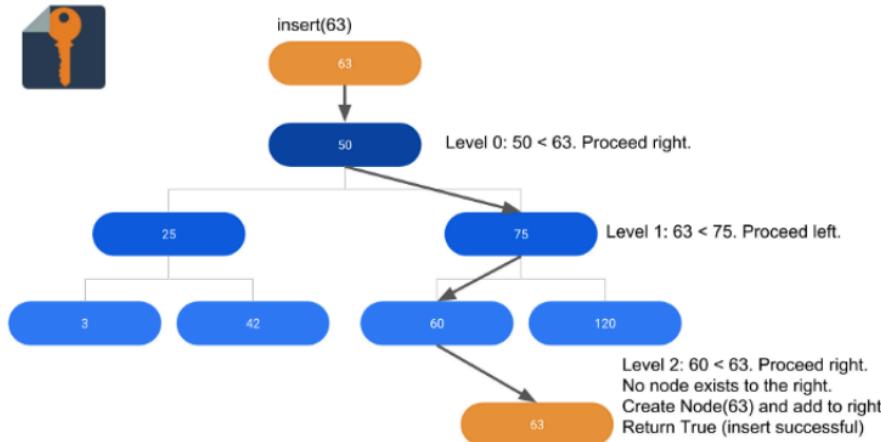
A sample BST

Binary Search Trees (BST)



Binary Search Trees (BST)

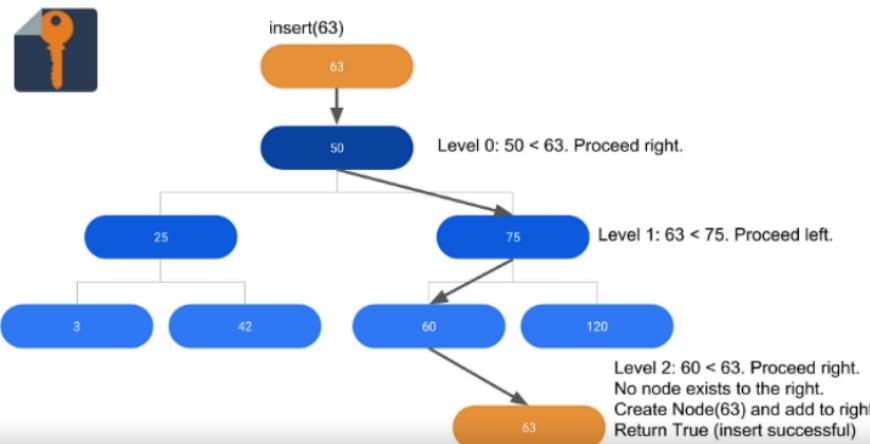
Insert Operation



```
class Node(object):
    ...
    def insert(self, d):
        if self.data == d:
            return False
        elif d < self.data:
            if self.left:
                return self.left.insert(d)
            else:
                self.left = Node(d)
                return True
        else:
            if self.right:
                return self.right.insert(d)
            else:
                self.right = Node(d)
                return True
    ...
}
```

Binary Search Trees (BST)

Insert Operation

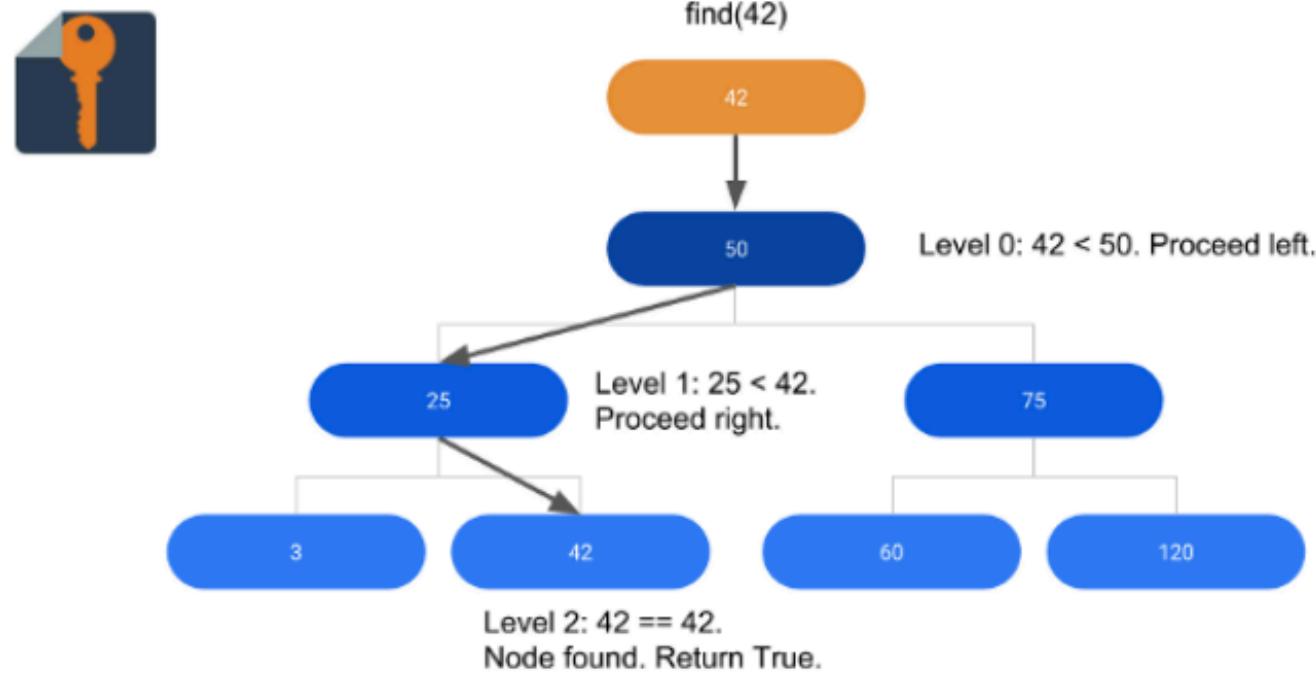


```
class BST(object):
    ...
    def insert(self, d):
        ...
        returns True if successfully inserted, false if exists
        ...
        if self.root:
            return self.root.insert(d)
        else:
            self.root = Node(d)
            return True
    ...
```

```
class Node(object):
    ...
    def insert(self, d):
        ...
        if self.data == d:
            return False
        elif d < self.data:
            if self.left:
                return self.left.insert(d)
            else:
                self.left = Node(d)
                return True
        else:
            if self.right:
                return self.right.insert(d)
            else:
                self.right = Node(d)
                return True
    ...
```

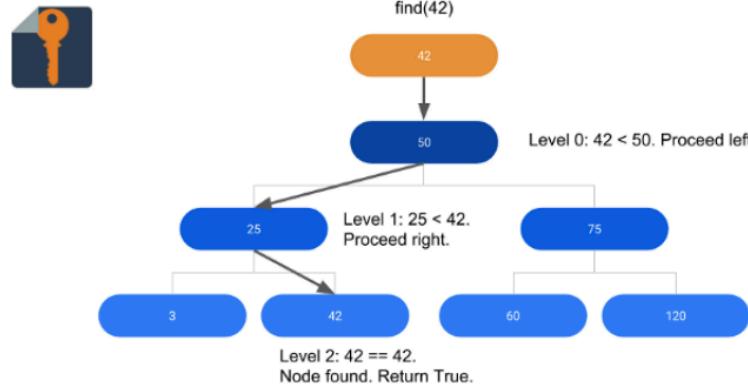
Binary Search Trees (BST)

Find (Search)Operation



Binary Search Trees (BST)

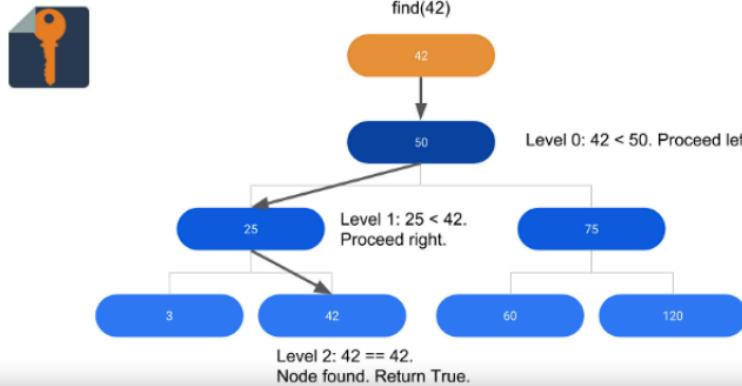
Find (Search)Operation



```
class Node(object):
    ...
    def find(self, d):
        if self.data == d:
            return True
        elif d < self.data and self.left:
            return self.left.find(d)
        elif d > self.data and self.right:
            return self.right.find(d)
        return False
    ...
```

Binary Search Trees (BST)

Find (Search)Operation



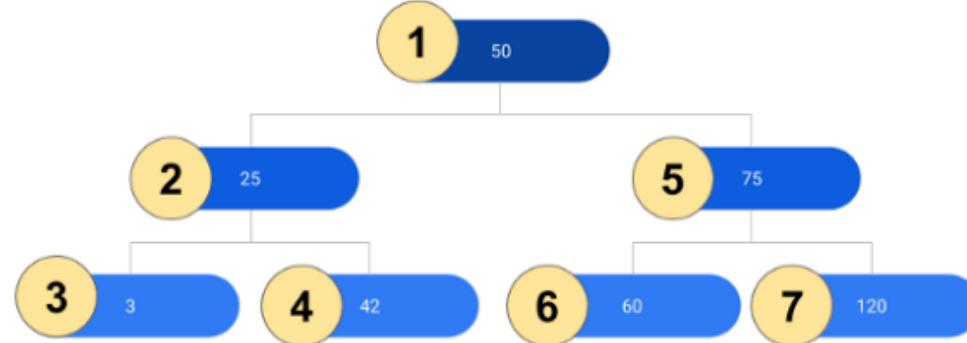
```
class Node(object):
...
def find(self, d):
    if self.data == d:
        return True
    elif d < self.data and self.left:
        return self.left.find(d)
    elif d > self.data and self.right:
        return self.right.find(d)
    return False
...
```

```
class BST(object):
...
def find(self, d):
    """
    Returns True if d is found in tree, false otherwise
    """
    if self.root:
        return self.root.find(d)
    else:
        return False
...
```

Tree Traversals

Preorder Traversal

root -> left -> right

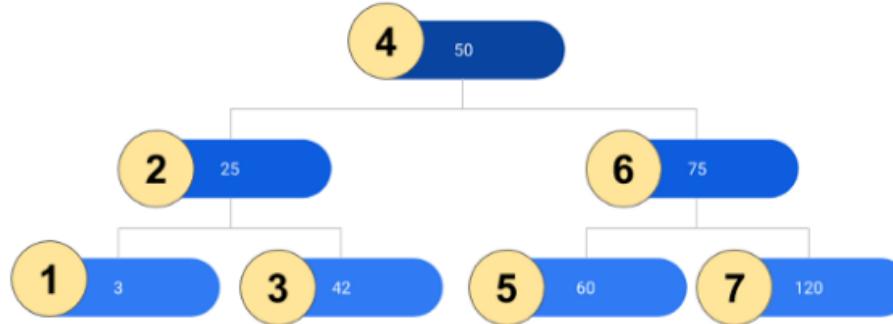


```
class Node(object):
    ...
    def preorder(self, l):
        ...
        l.append(self.data)
        if self.left:
            self.left.preorder(l)
        if self.right:
            self.right.preorder(l)
        return l
    ...
```

Tree Traversals

Inorder Traversal

left -> root -> right

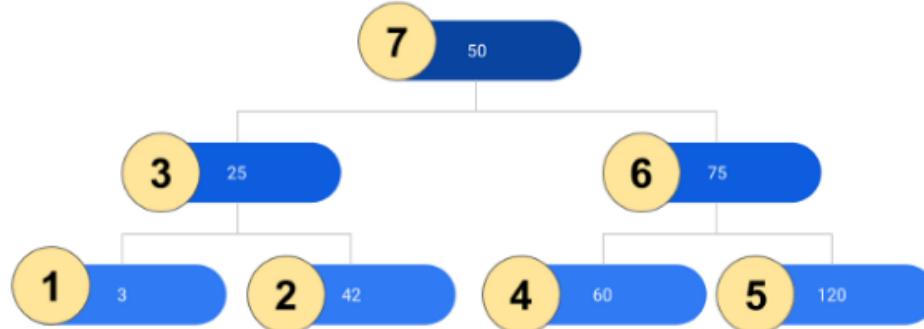


```
class Node(object):
    ...
    def inorder(self, l):
        ...
        l: the list of data objects so far in the traversal
        ...
        if self.left:
            self.left.inorder(l)
        l.append(self.data)
        if self.right:
            self.right.inorder(l)
        return l
    ...
```

Tree Traversals

Postorder Traversal

left -> right -> root



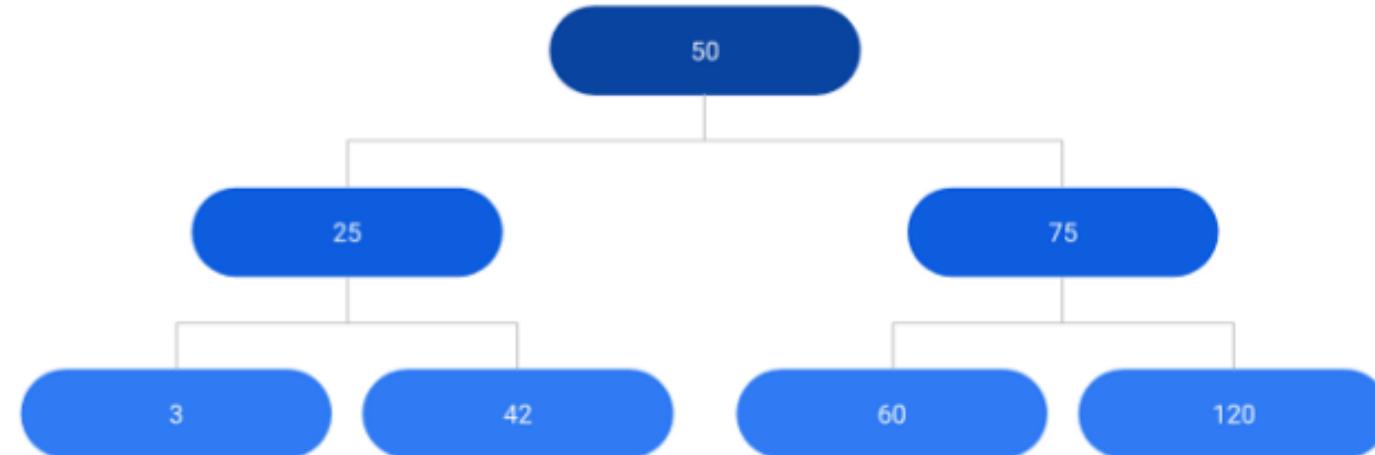
```
class Node(object):
    ...
    def postorder(self, l):
        ...
        l: the list of data objects so far in the traversal
        ...
        if self.left:
            self.left.postorder(l)
        if self.right:
            self.right.postorder(l)
        l.append(self.data)
        return l
    ...
```

BST Traversals

```
class BST(object):
    def preorder(self):
        """
        Returns list of data elements resulting from preorder tree
        traversal
        """
        if self.root:
            return self.root.preorder([])
        else:
            return []
    def postorder(self):
        """
        Returns list of post-order elements
        """
        if self.root:
            return self.root.postorder([])
        else:
            return []
    def inorder(self):
        """
        Returns list of in-order elements
        """
        if self.root:
            return self.root.inorder([])
        else:
            return []
```

BST Traversals

The inorder traversal of this BST is ...



BST Traversals

The inorder traversal of this BST is ...



[3, 25, 42, 50, 60, 75, 120]

Sorted List of elements

Lecture 26

ADT - Linked Lists & Queues

Dr. Hacer Yalım Keleş

hkeles@ankara.edu.tr

Disclaimer: *Most of the slide contents are adapted from the Book: Problem Solving with Algorithms and Data Structures using Python*

Linked Lists

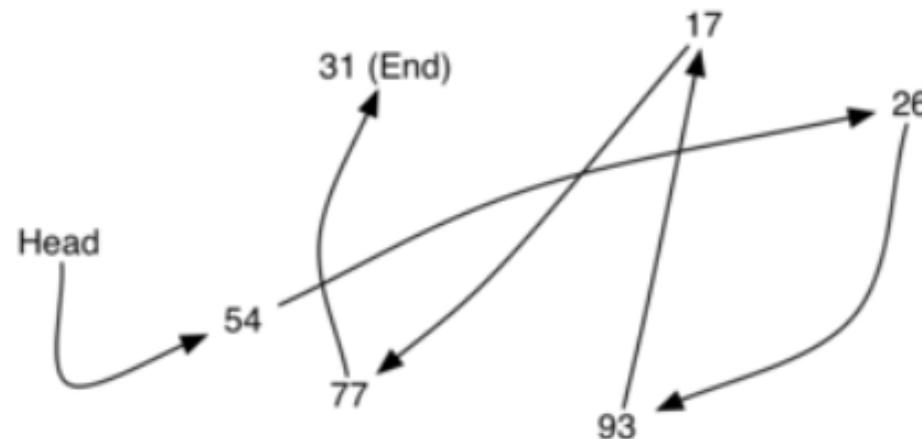
There is no requirement that we maintain that positioning in contiguous memory



Linked Lists

The location of the first item of the list must be explicitly specified.

Once we know where the first item is, the first item can tell us where the second is, and so on.



The external reference is often referred to as the **head** of the list.

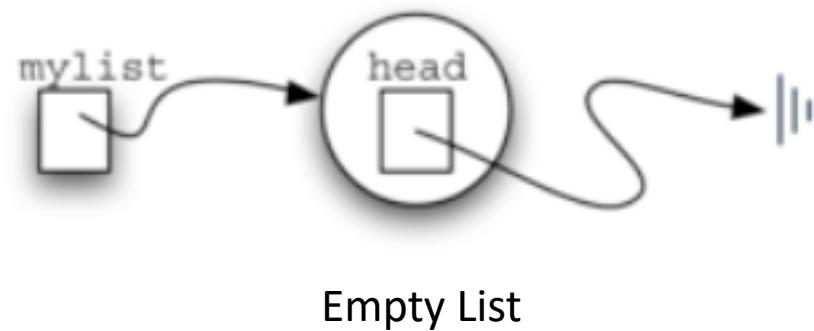
Node Class

```
class Node:  
    def __init__(self,initdata):  
        self.data = initdata  
        self.next = None  
  
    def getData(self):  
        return self.data  
  
    def getNext(self):  
        return self.next  
  
    def setData(self,newdata):  
        self.data = newdata  
  
    def setNext(self,newnext):  
        self.next = newnext
```

Unordered Linked List Class

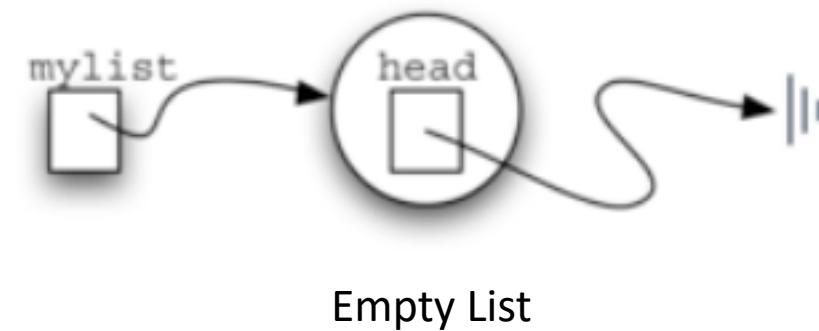
```
class UnorderedList:  
  
    def __init__(self):  
        self.head = None
```

```
>>> mylist = UnorderedList()
```



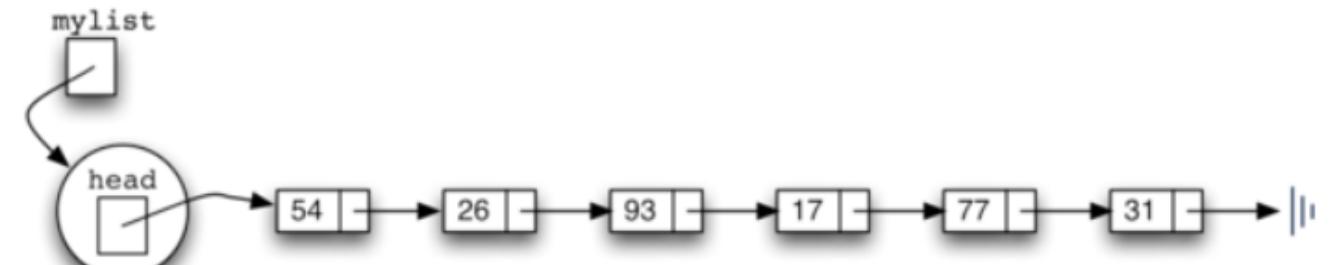
Unordered List Class

```
class UnorderedList:  
  
    def __init__(self):  
        self.head = None
```



```
>>> mylist = UnorderedList()
```

```
def isEmpty(self):  
    return self.head == None
```

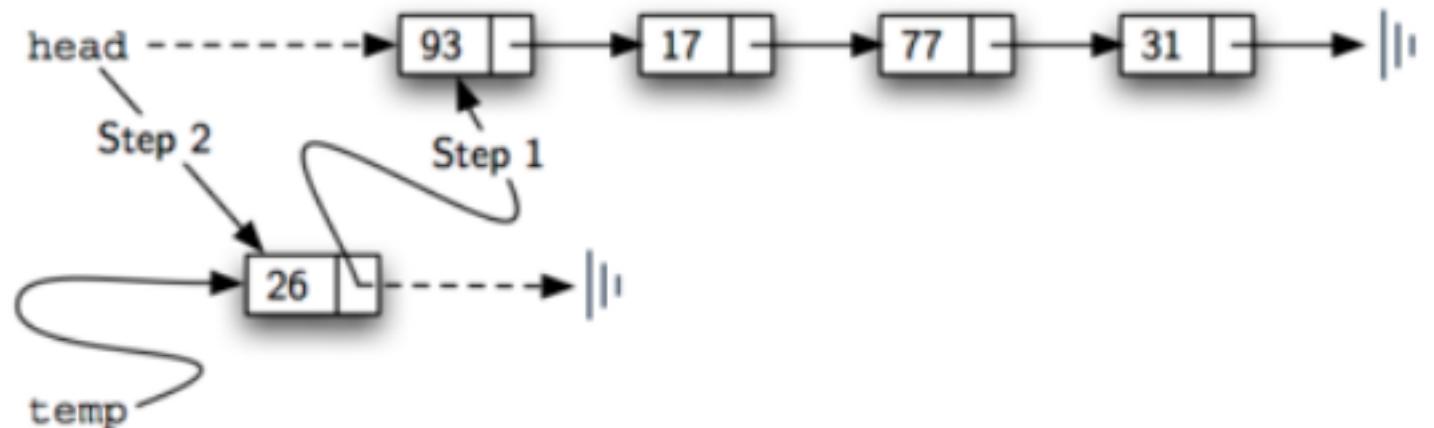


A Linked List of Integers

Unordered List Class

```
class UnorderedList:  
    def __init__(self):  
        self.head = None
```

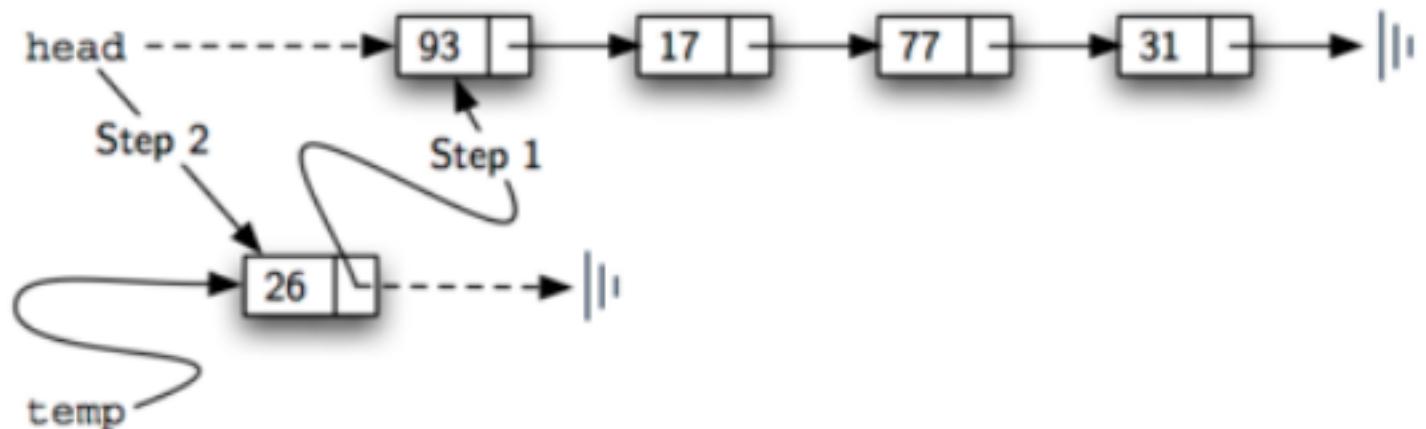
Adding an item to the linked list



Unordered List Class

```
class UnorderedList:  
  
    def __init__(self):  
        self.head = None  
  
    def add(self, item):  
        temp = Node(item)  
        temp.setNext(self.head)  
        self.head = temp
```

Adding an item to the linked list



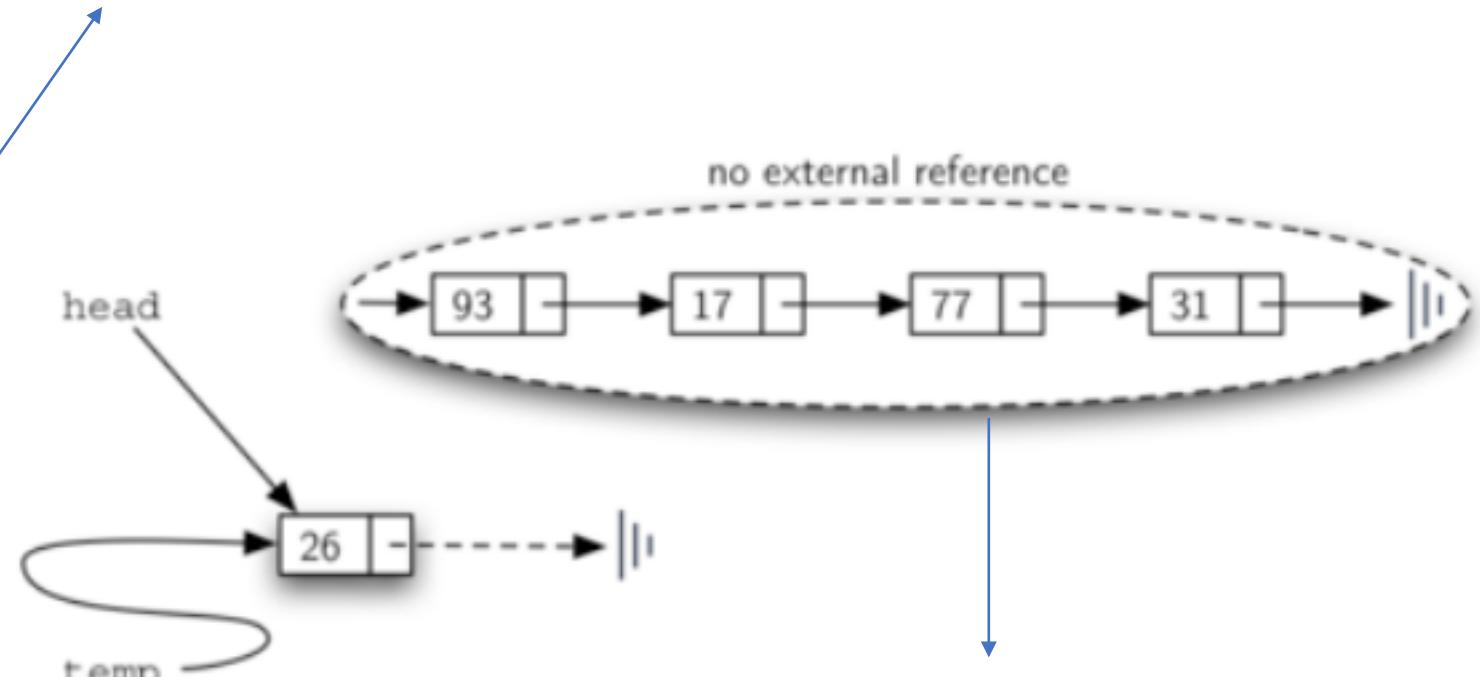
Unordered List Class

```
class UnorderedList:  
  
    def __init__(self):  
        self.head = None
```

```
def add(self, item):  
    temp = Node(item)  
    temp.setNext(self.head)  
    self.head = temp
```

Order is important!

What if we first set the head of the list?



We would lose the rest of the list

Unordered List Class

```
class UnorderedList:

    def __init__(self):
        self.head = None

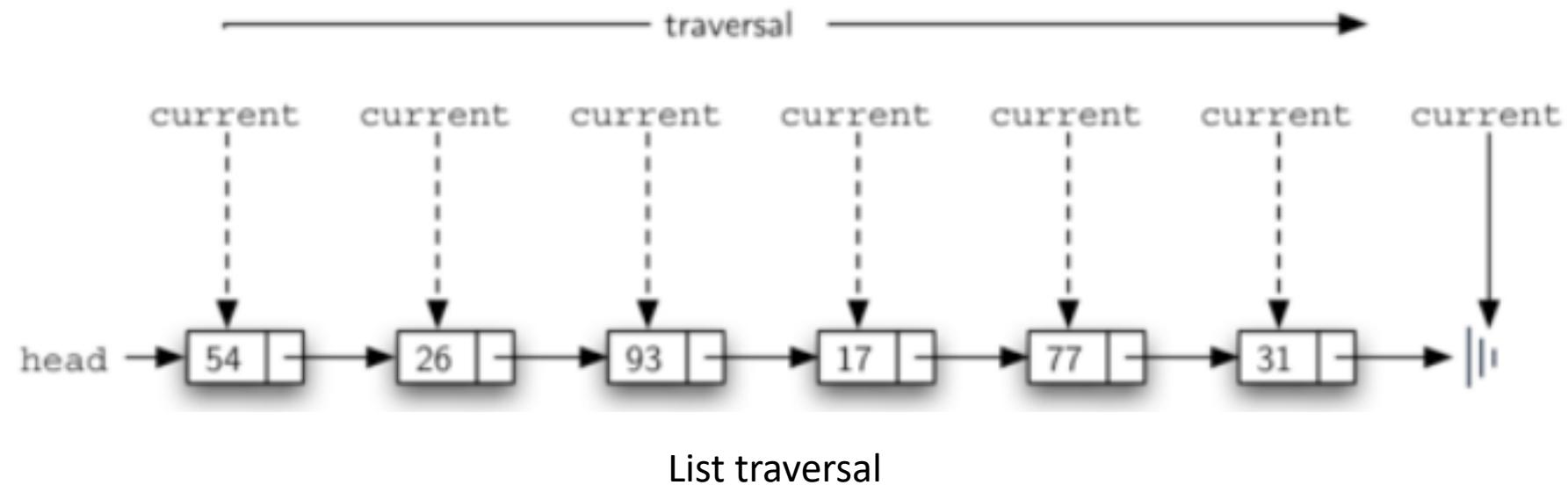
    def add(self, item):
        temp = Node(item)
        temp.setNext(self.head)
        self.head = temp
```

```
>>> mylist.add(31)
>>> mylist.add(77)
>>> mylist.add(17)
>>> mylist.add(93)
>>> mylist.add(26)
>>> mylist.add(54)
```



Unordered List Class

```
class UnorderedList:  
  
    def __init__(self):  
        self.head = None
```



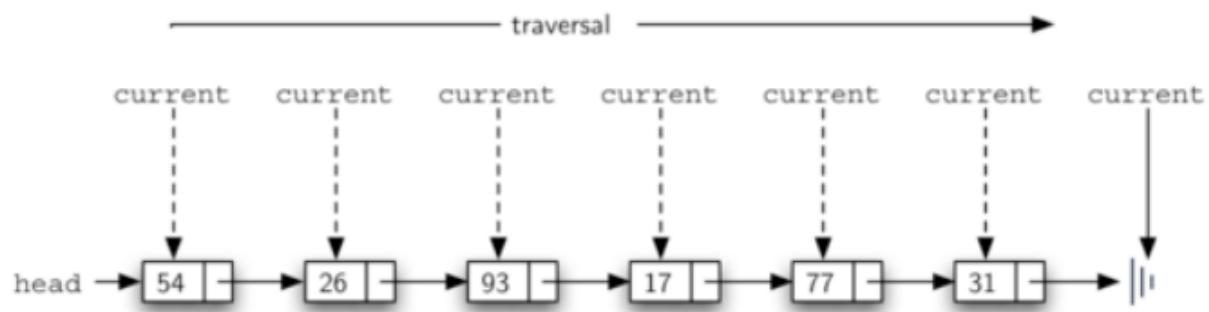
Unordered List Class

```
class UnorderedList:

    def __init__(self):
        self.head = None

    def size(self):
        current = self.head
        count = 0
        while current != None:
            count = count + 1
            current = current.getNext()

    return count
```



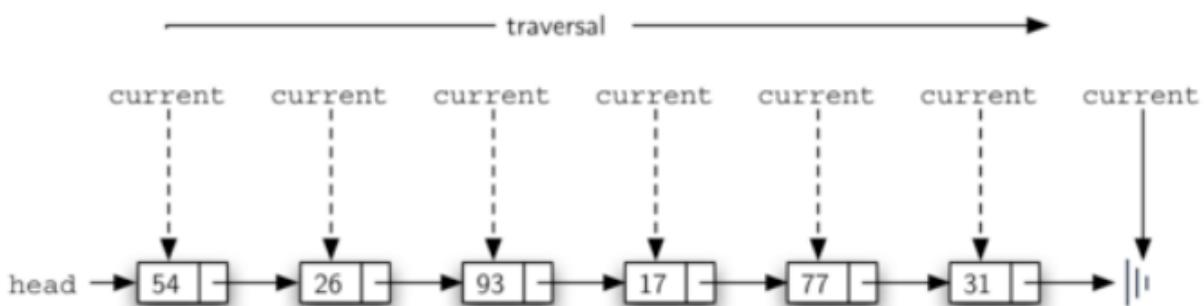
Unordered List Class

```
class UnorderedList:

    def __init__(self):
        self.head = None

    def search(self, item):
        current = self.head
        found = False
        while current != None and not found:
            if current.getData() == item:
                found = True
            else:
                current = current.getNext()

        return found
```

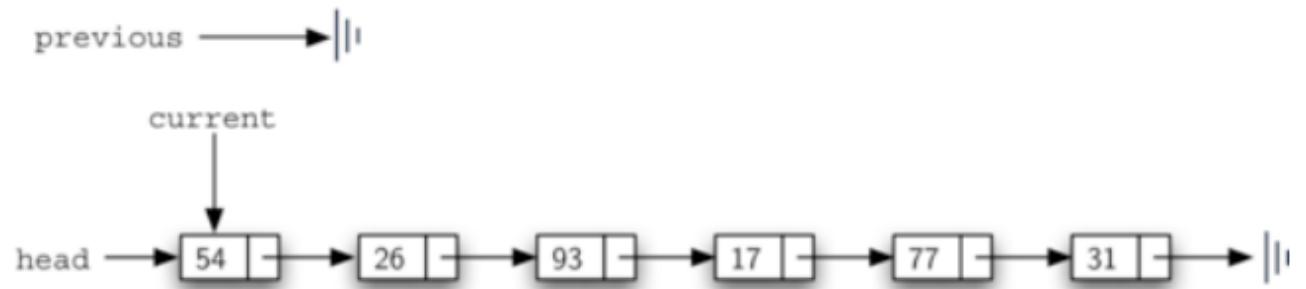


```
>>> myList.search(17)
True
```

Unordered List Class

```
def remove(self, item):
    current = self.head
    previous = None
    found = False
    while not found:
        if current.getData() == item:
            found = True
        else:
            previous = current
            current = current.getNext()

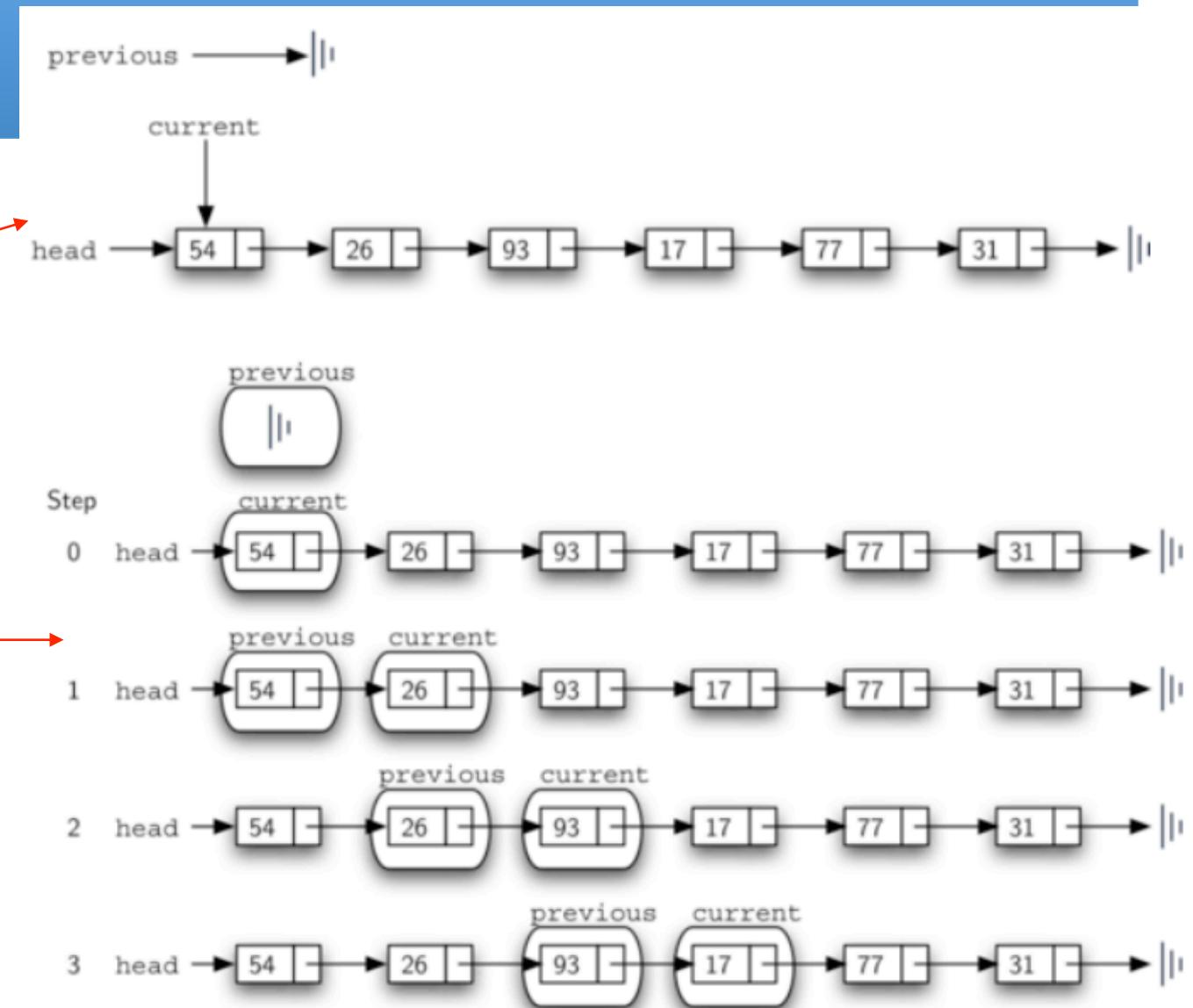
    if previous == None:
        self.head = current.getNext()
    else:
        previous.setNext(current.getNext())
```



Unordered List Class

```
def remove(self, item):
    current = self.head
    previous = None
    found = False
    while not found:
        if current.getData() == item:
            found = True
        else:
            previous = current
            current = current.getNext()

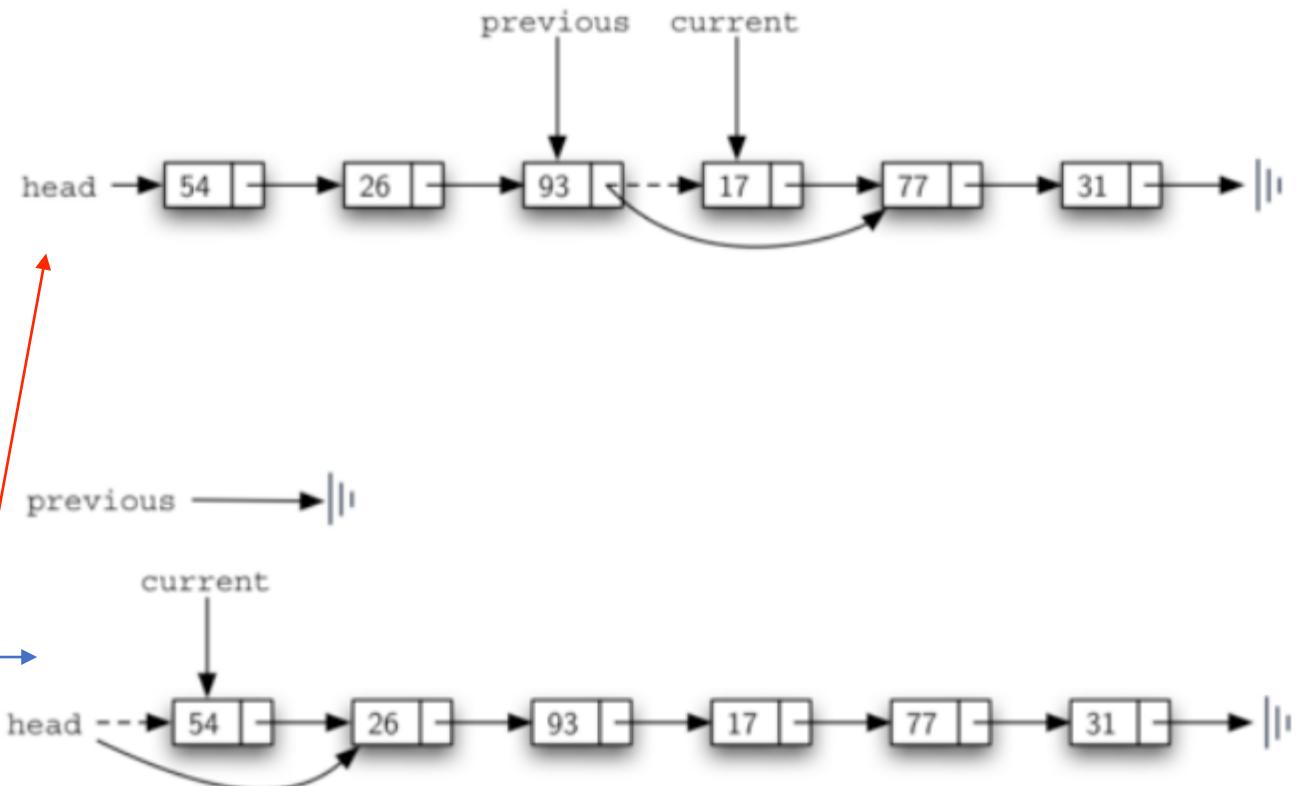
    if previous == None:
        self.head = current.getNext()
    else:
        previous.setNext(current.getNext())
```



Unordered List Class

```
def remove(self, item):
    current = self.head
    previous = None
    found = False
    while not found:
        if current.getData() == item:
            found = True
        else:
            previous = current
            current = current.getNext()

    if previous == None:
        self.head = current.getNext()
    else:
        previous.setNext(current.getNext())
```



Ordered Linked List Class



```
class OrderedList:  
    def __init__(self):  
        self.head = None
```

Ordered Linked List Class

```
class OrderedList:  
    def __init__(self):  
        self.head = None
```

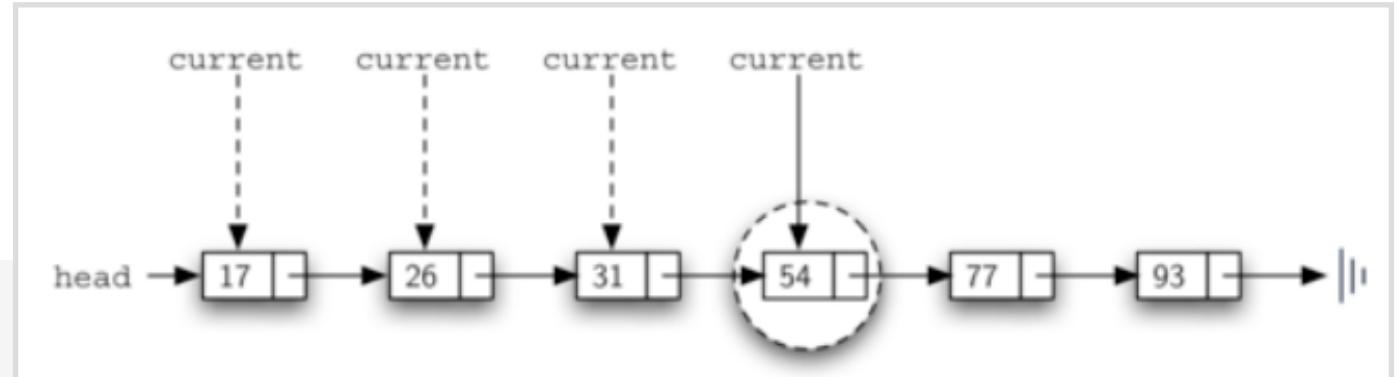


isEmpty, size and remove methods are the same with the [unordered list class](#)

add and search methods will be modified

Ordered Linked List Class

```
class OrderedList:  
    def __init__(self):  
        self.head = None  
  
    def search(self, item):  
        current = self.head  
        found = False  
        stop = False  
        while current != None and not found and not stop:  
            if current.getData() == item:  
                found = True  
            else:  
                if current.getData() > item:  
                    stop = True  
                else:  
                    current = current.getNext()  
  
        return found
```



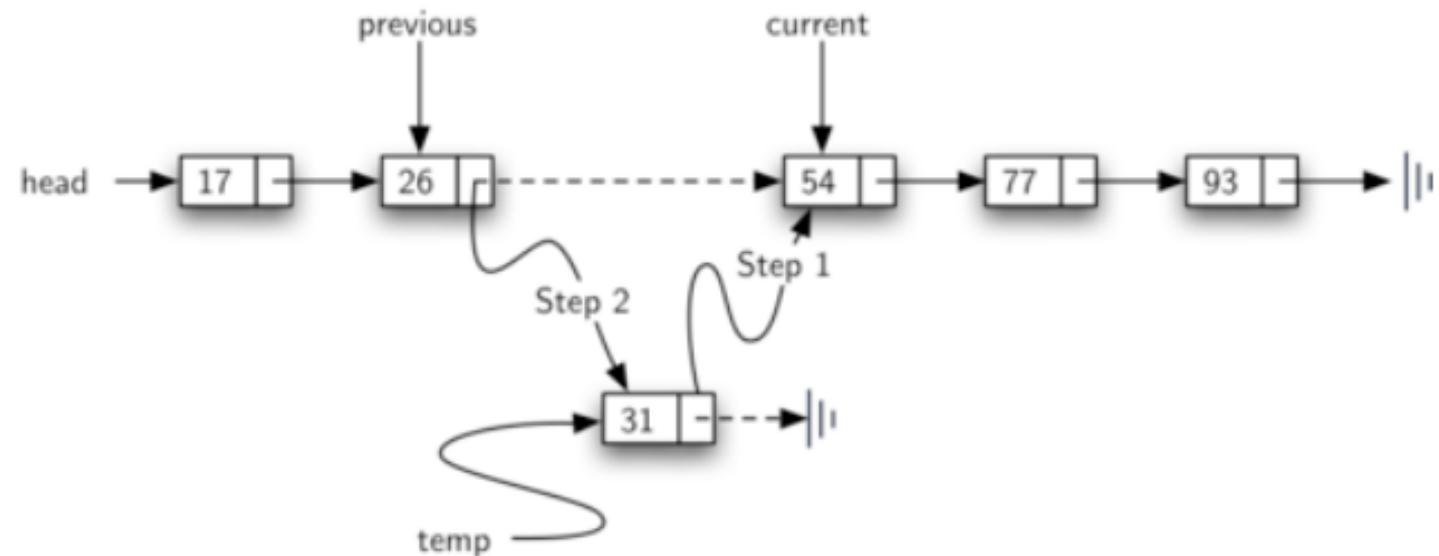
Use the advantage of the order to early stop

If any node is ever discovered that contains data greater than the item we are looking for, we will set `stop` to `True`

Ordered Linked List Class

add method

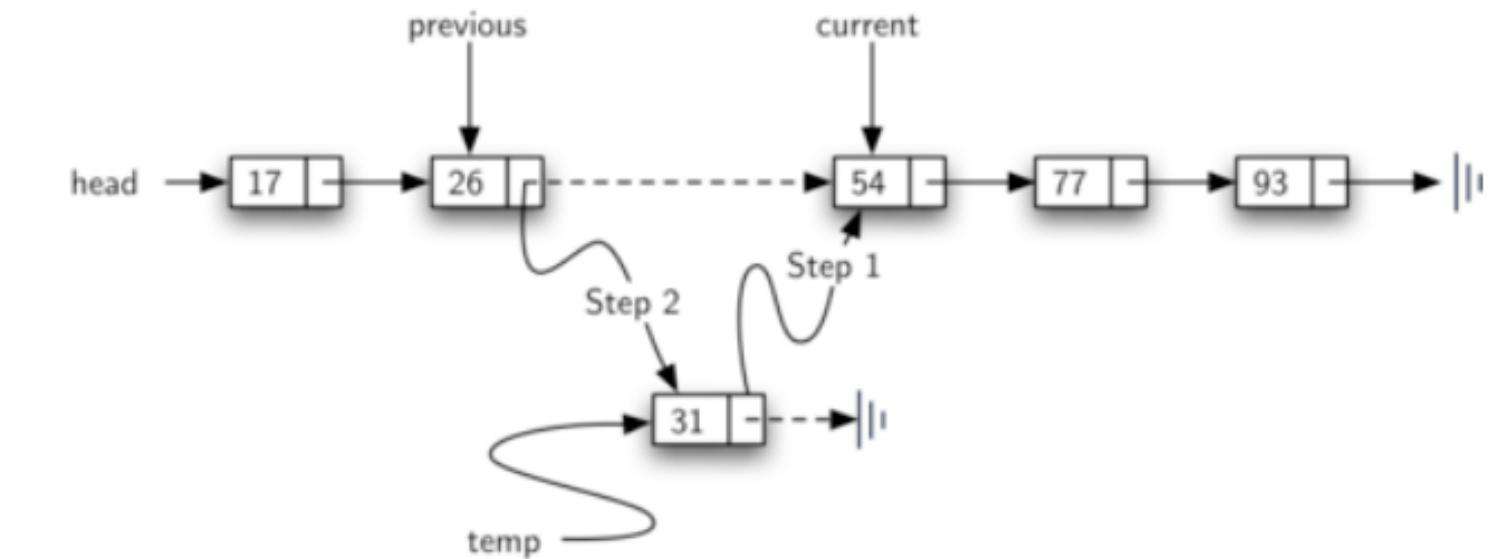
Assume we have the ordered list consisting of 17, 26, 54, 77, and 93 and we want to add the value 31.



Ordered Linked List Class

```
def add(self, item):
    current = self.head
    previous = None
    stop = False
    while current != None and not stop:
        if current.getData() > item:
            stop = True
        else:
            previous = current
            current = current.getNext()

    temp = Node(item)
    if previous == None:
        temp.setNext(self.head)
        self.head = temp
    else:
        temp.setNext(current)
        previous.setNext(temp)
```



Consider adding 10 to this list

What is a Queue?

FIFO: First In First Out



The simplest example of a queue is the typical line that we all participate in from time to time.

We wait in a line for a movie, we wait in the check-out line at a grocery store, and we wait in the cafeteria line

What is a Queue ADT?

A queue is structured as an ordered collection of items which are added at one end, called the “[rear](#),” and removed from the other end, called the “[front](#).”

[Queue\(\)](#) creates a new queue that is empty. It needs no parameters and returns an empty queue.

[enqueue\(item\)](#) adds a new item to the [rear](#) of the queue. It needs the item and returns nothing.

[dequeue\(\)](#) removes the [front item](#) from the queue. It needs no parameters and returns the item. The queue is modified.

[isEmpty\(\)](#) tests to see whether the queue is empty. It needs no parameters and returns a boolean value.

[size\(\)](#) returns the number of items in the queue. It needs no parameters and returns an integer.

What is a Queue ADT?

A queue is structured as an ordered collection of items which are added at one end, called the “[rear](#),” and removed from the other end, called the “[front](#).”

[Queue\(\)](#) creates a new queue that is empty. It needs no parameters and returns an empty queue.

[enqueue\(item\)](#) adds a new item to the [rear](#) of the queue. It needs the item and returns nothing.

[dequeue\(\)](#) removes the [front item](#) from the queue. It needs no parameters and returns the item. The queue is modified.

[isEmpty\(\)](#) tests to see whether the queue is empty. It needs no parameters and returns a boolean value.

[size\(\)](#) returns the number of items in the queue. It needs no parameters and returns an integer.

Queue Operation	Queue Contents	Return Value
<code>q.isEmpty()</code>	[]	True
<code>q.enqueue(4)</code>	[4]	
<code>q.enqueue('dog')</code>	['dog', 4]	
<code>q.enqueue(True)</code>	[True, 'dog', 4]	
<code>q.size()</code>	[True, 'dog', 4]	3
<code>q.isEmpty()</code>	[True, 'dog', 4]	False
<code>q.enqueue(8.4)</code>	[8.4, True, 'dog', 4]	
<code>q.dequeue()</code>	[8.4, True, 'dog']	4
<code>q.dequeue()</code>	[8.4, True]	'dog'
<code>q.size()</code>	[8.4, True]	2

Implementation of a Queue class

the rear is at position 0 in the list.

This allows us to use the `insert` function on lists to add new elements to the rear of the queue.

The `pop` operation can be used to remove the front element (the last element of the list)

```
class Queue:  
    def __init__(self):  
        self.items = []  
  
    def isEmpty(self):  
        return self.items == []  
  
    def enqueue(self, item):  
        self.items.insert(0, item)  
  
    def dequeue(self):  
        return self.items.pop()  
  
    def size(self):  
        return len(self.items)
```