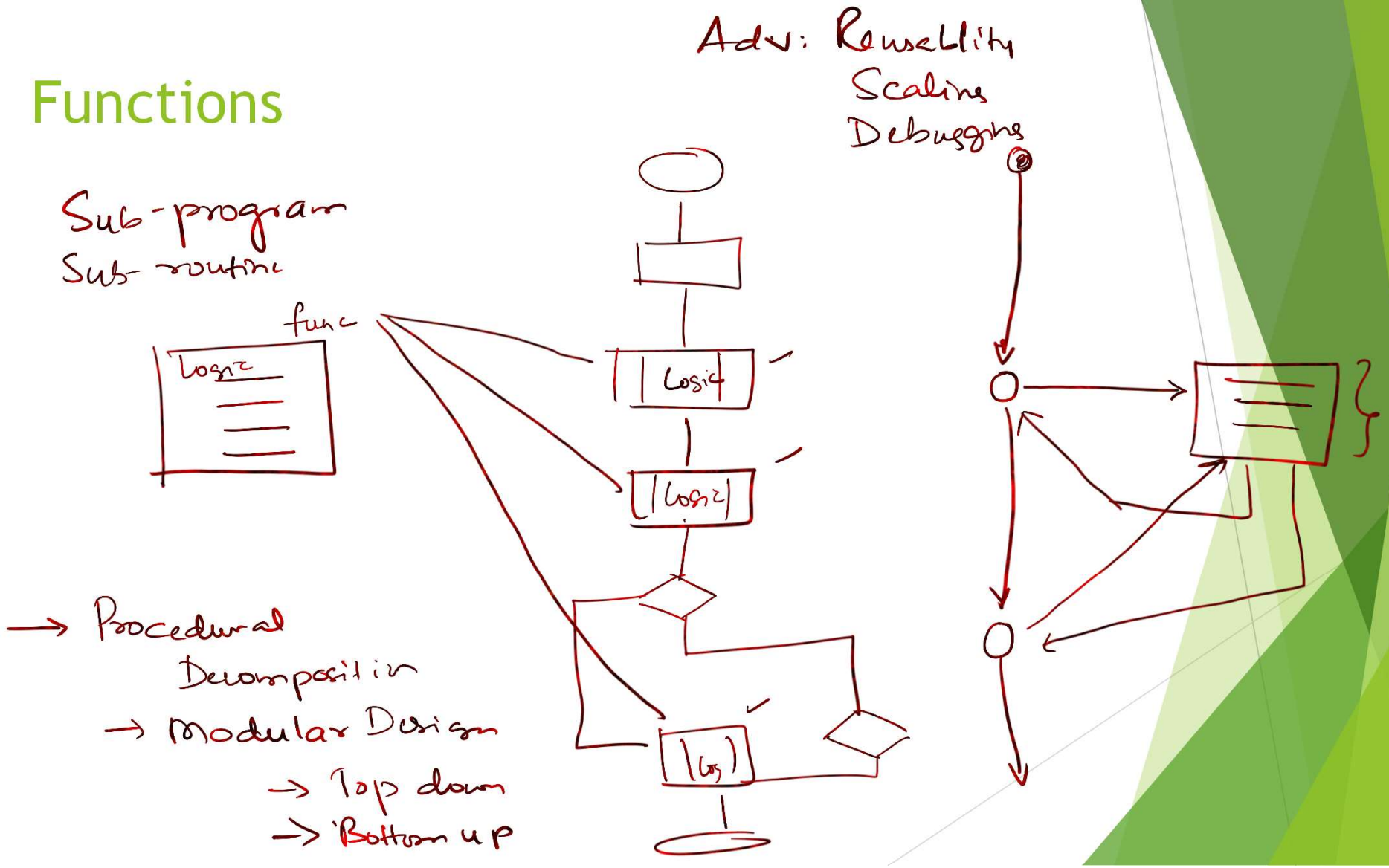
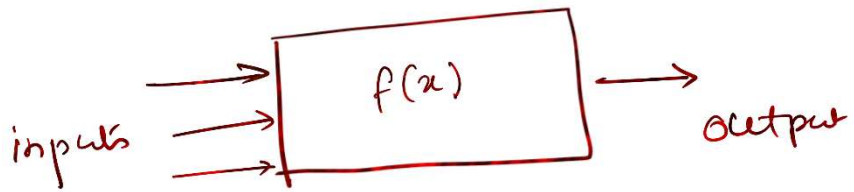
Functions


Adv: Reusability
Scaling
Debugging
Sub-program
Sub-routine
func
Logic
Logic
Logic
Logic
→ Procedural
Decomposition
→ Modular Design
→ Top down
→ Bottom up

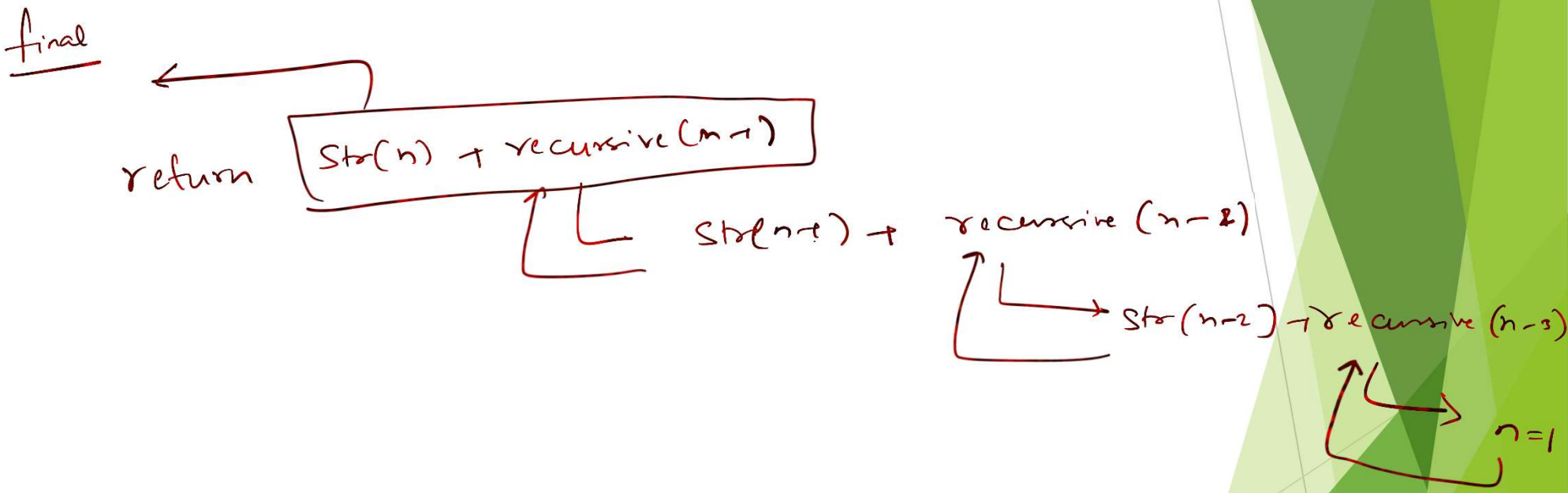
Function



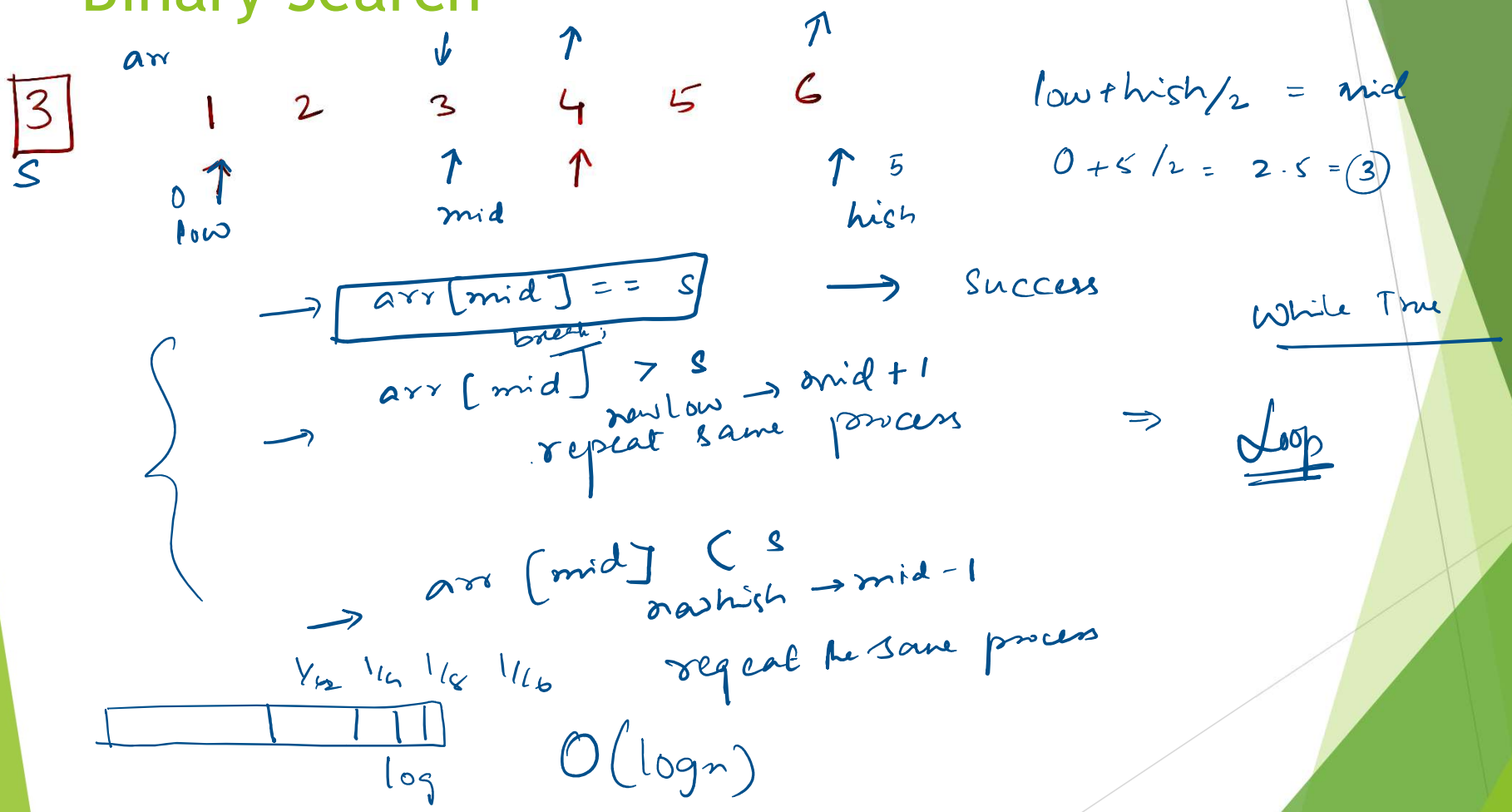

```
def <function_name> ( <inputs> ):
    <logic>
    return <output>
```

Recursive Functions

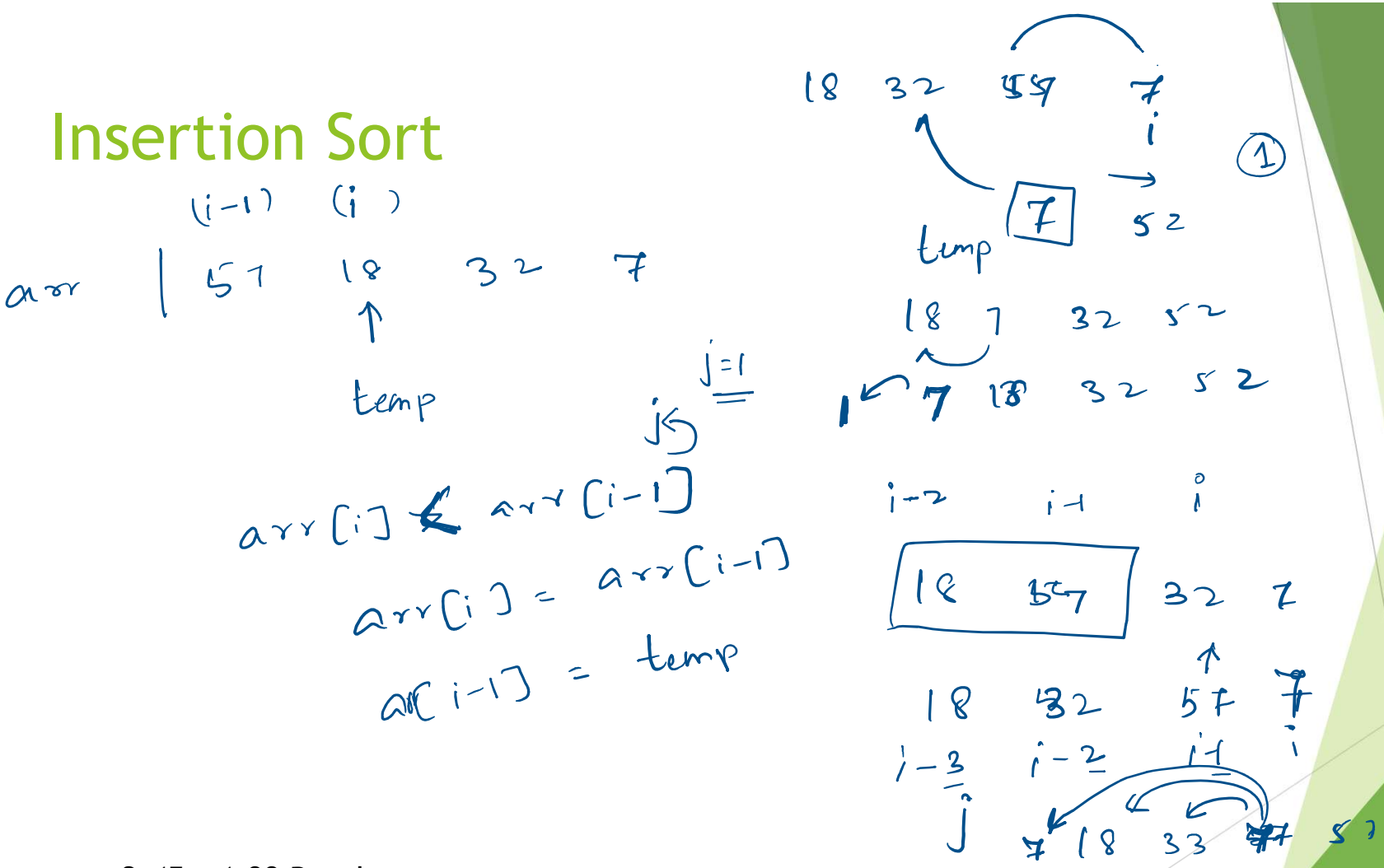
1:30 – 2:00 Lunch Time
2:00 – 2:30 Quick Test



Binary Search

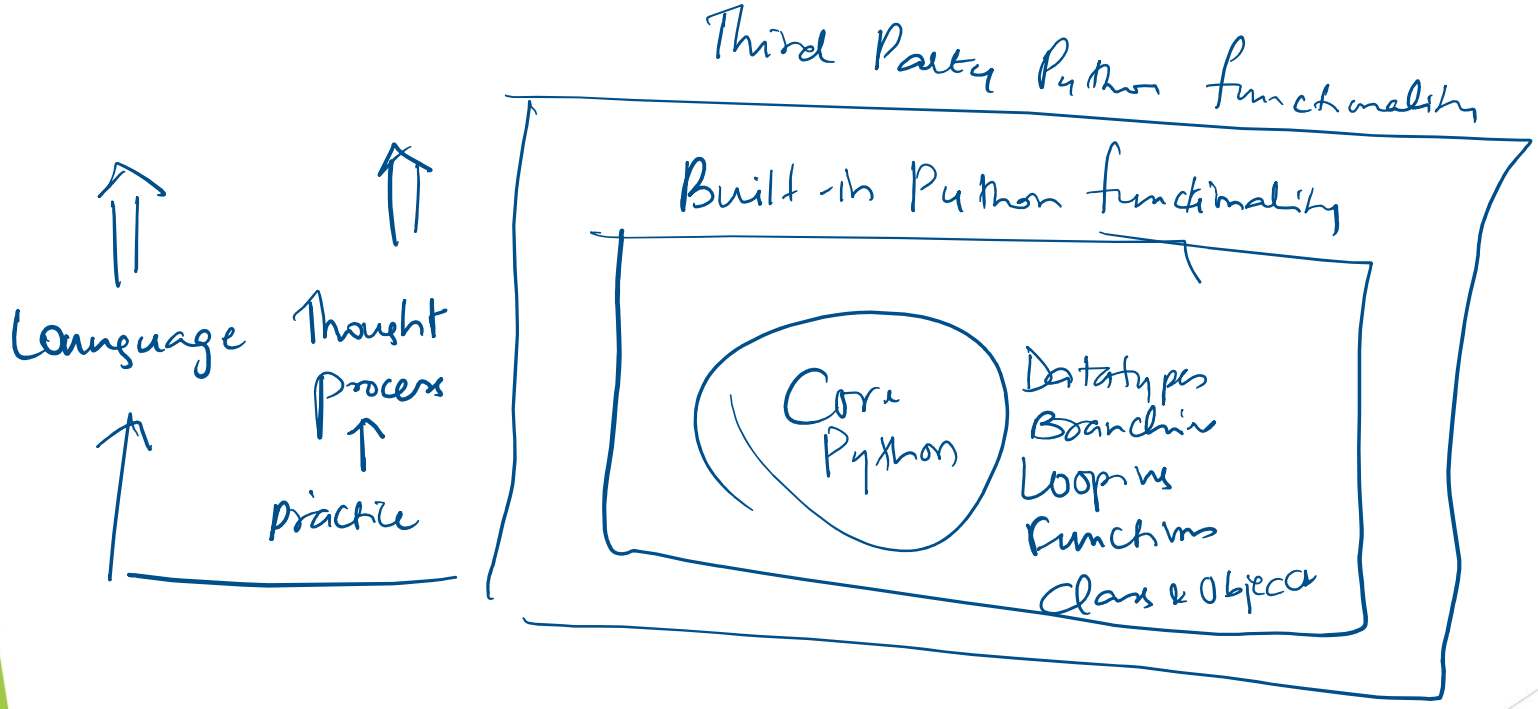


Insertion Sort



3:45 – 4:00 Break
4:00 – 4:30 Leadership Connect

Understanding Python



File System and Directories

- Access to your file system occurs mostly through the Python **os** module
- **os** module is actually a front-end to the real module that is loaded, a module that is clearly operating system dependent
- This "real" module may be one of the following: posix (Unix-based, i.e., Linux, MacOS X, *BSD, Solaris, etc.), nt (Win32), mac (old MacOS), dos (DOS), os2 (OS/2), etc.
- You should never import those modules directly
- Just import os and the appropriate module will be loaded, keeping all the underlying work hidden from sight

Refer:
https://docs.python.org/3/library/os.html
https://docs.python.org/2/library/os.html

Important Functions in **os** Module

Function	Purpose
mkdir()/makedirs()	Create directory(ies)
rmdir()/removedirs()	Remove directory(ies)
getcwd()/getcwdu()	Return current working directory/same but in Unicode
listdir()	List files in directory
chroot()	Change root direcory of current process
chdir()/fchdir()	Change working directory/via a file descriptor
access()	Verify permission modes
chmod()	Change permission modes
remove()/unlink()	Delete file
rename()/renames()	Rename file
open()	Low-level operating system open [for files, use the **standard open() built-in** functions
read()/write()	Read/write data to a file descriptor

os.path Module

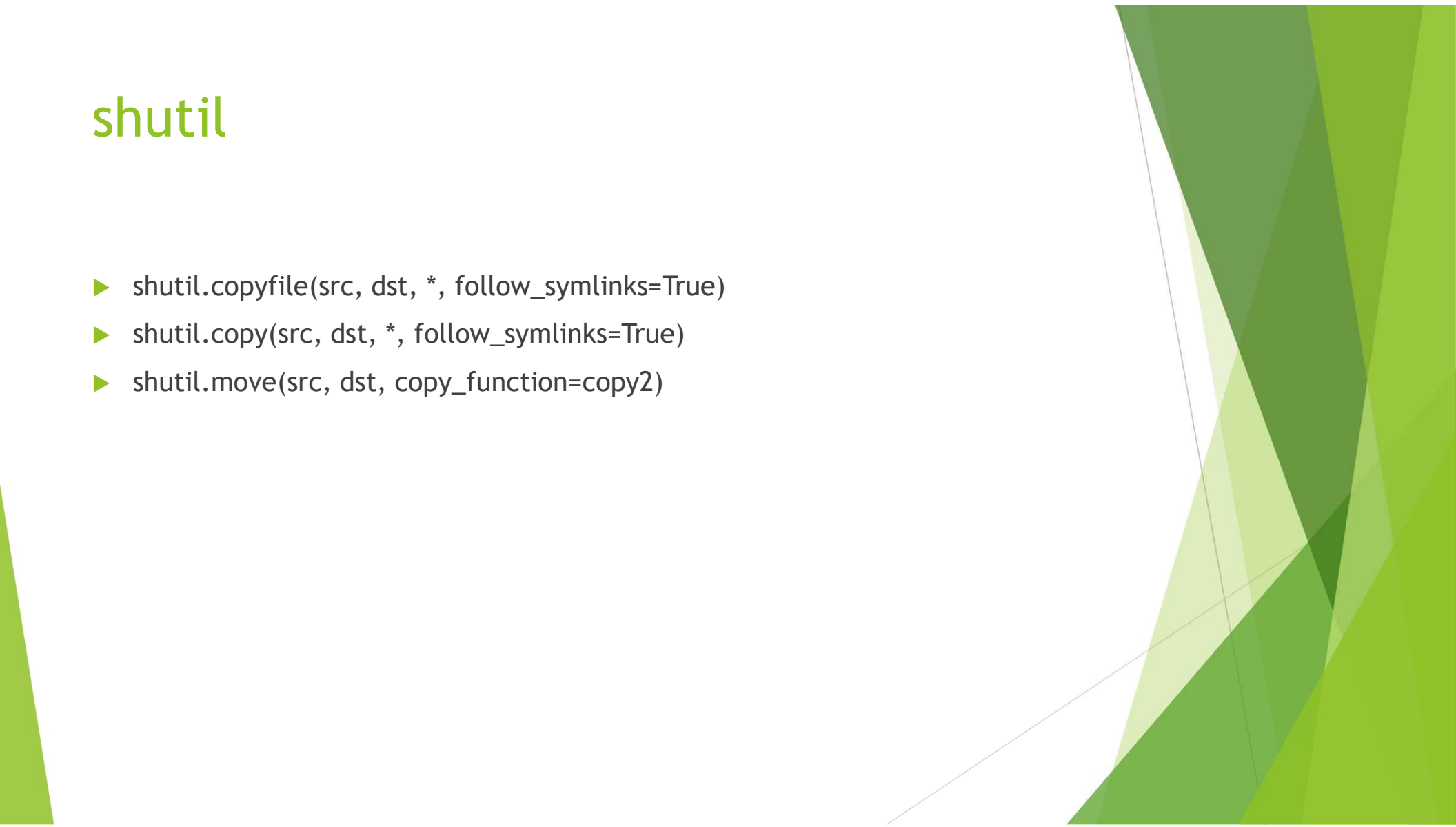
- A second module that performs specific pathname operations is also available which called **os.path**
- The **os.path** module is accessible through the **os** module
- Included with this module are functions to manage and manipulate file pathname components, obtain file or directory information, and make file path inquiries

Important Functions in **os.path** Module

Function	Purpose
basename()	Remove directory path and return leaf name
dirname()	Remove leaf name and return directory path
join()	Join separate components into single pathname
split()	Return (dirname(), basename()) tuple
splitdrive()	Return (drivename, pathname) tuple
splitext()	Return (filename, extension) tuple
getatime()	Return last file access time
getctime()	Return file creation time
getmtime()	Return last file modification time
getsize()	Return file size (in bytes)
exists()	Does pathname (file or directory) exist?
isabs()	Is pathname absolute?
isdir()	Does pathname exist and is a directory?
isfile()	Does pathname exist and is a file?
islink()	Does pathname exist and is a symbolic link?
ismount()	Does pathname exist and is a mount point?
samefile()	Do both pathnames point to the same file?

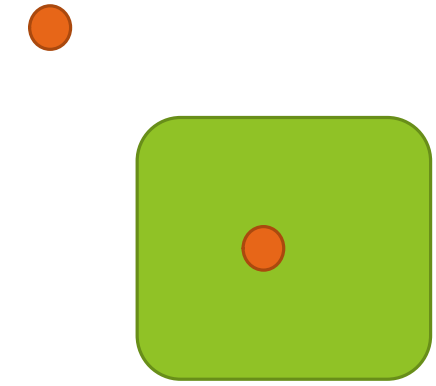
shutil

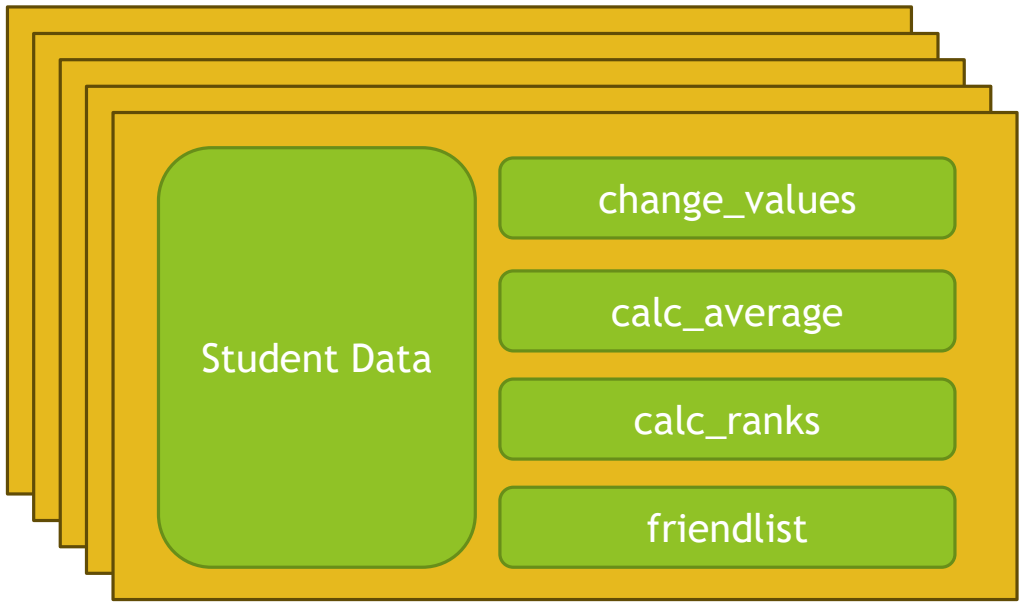
- shutil.copyfile(src, dst, *, follow_symlinks=True)
- shutil.copy(src, dst, *, follow_symlinks=True)
- shutil.move(src, dst, copy_function=copy2)



OOP

- Why Object Oriented Programming?
- Key aspects:
  - Reusability
  - Enhancements and upgradability, scalability
  - Modularity
  - Testability
  - Usability
  - Redundancy
  - Protection of data from accidental changes
  - Maintainability





calc_tax

setmaritalstatus

getchildren

getbankbalance

getlicensedata

getcriminalrecord

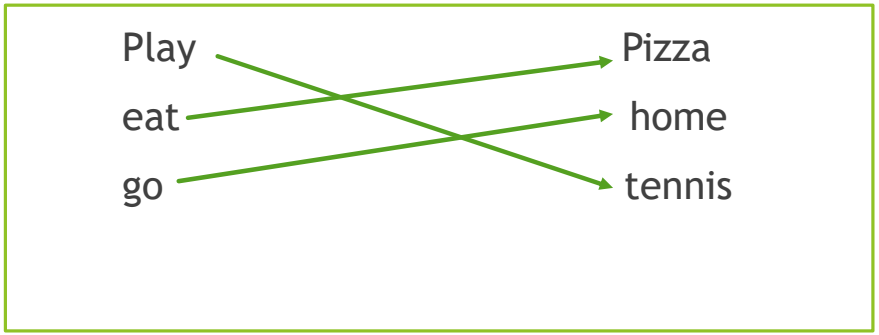
U

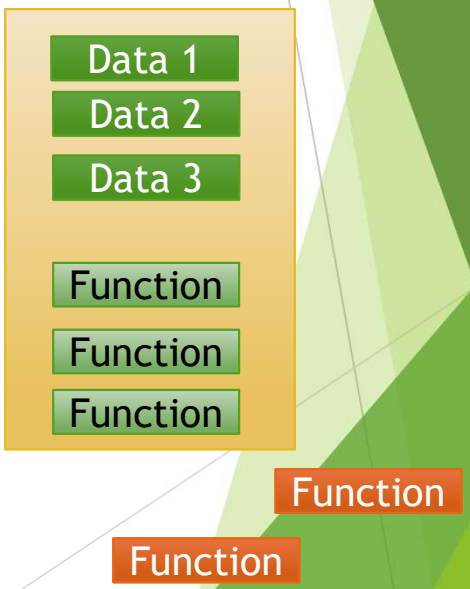
D

But why “object-oriented” ness in programming

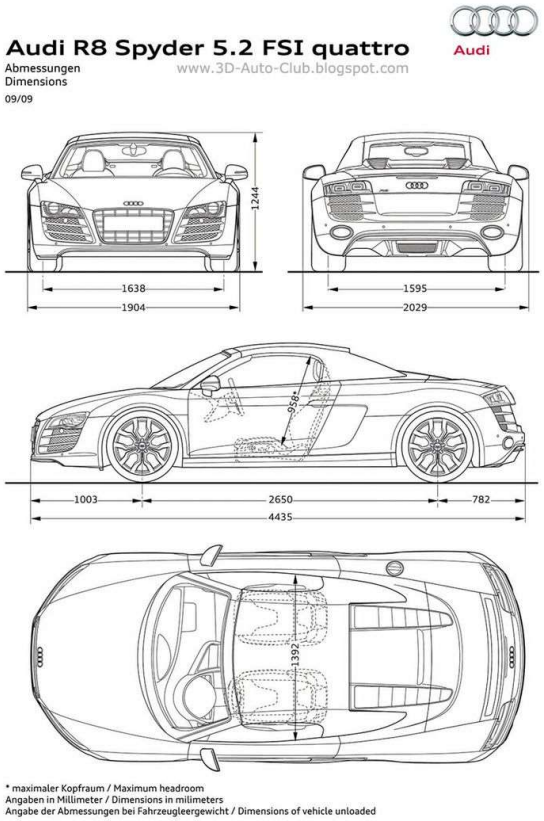
- Combine/associate the data and relevant functions in a single entity

Ram





Understanding a class



Object

- An object is a instance of a class and its an entity which has its own set of attributes and functions that were defined by a class

Public Interface

- The set of all methods provided by a class, together with the description of their behavior is called the public interface of the class

Encapsulation

- Encapsulation is the act of providing a public interface and hiding the implementation details
- Encapsulation enables changes in the implementation without affecting users of the class

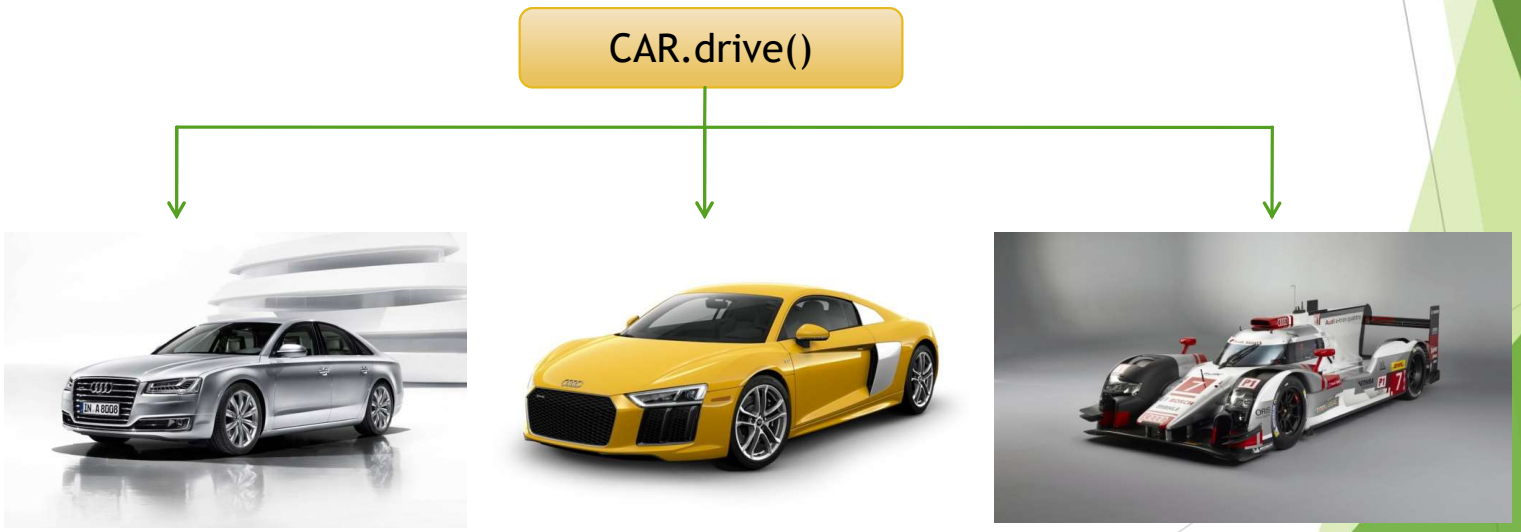
You can drive a car by operating the steering wheel and pedals, without knowing how the engine works. Similarly, you use an object through its methods. The implementation is hidden.

Inheritance

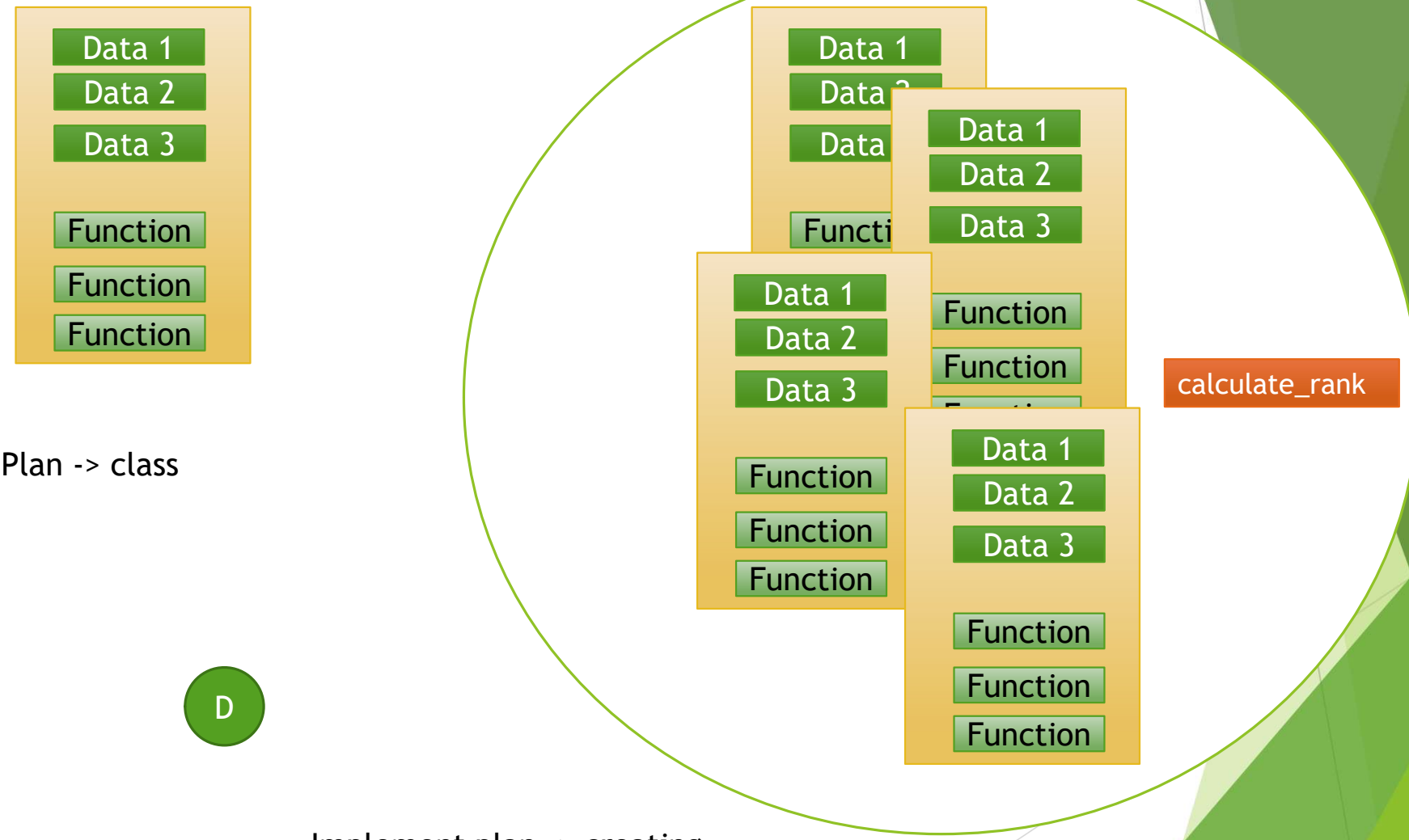
- Inheritance is a way to form new classes and thereafter objects using classes that have already been defined.

Polymorphism

- Polymorphism means that meaning of operation depends on the object being operated on.



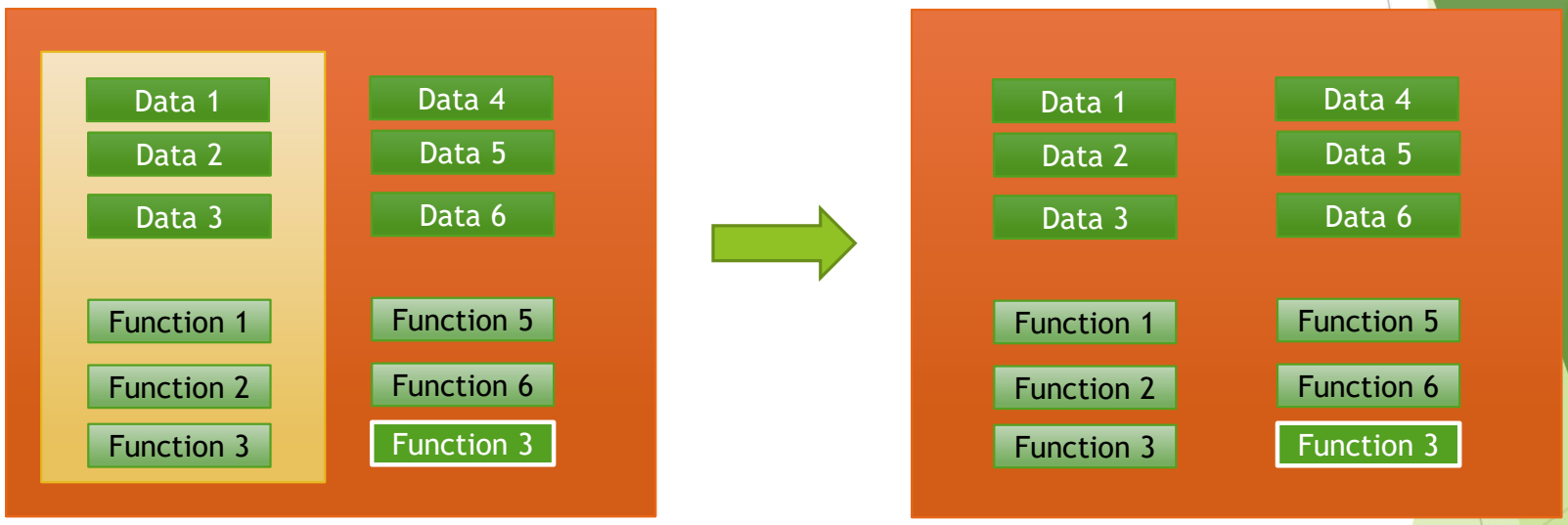
Let’s discuss about driving and maintenance tasks for cars…



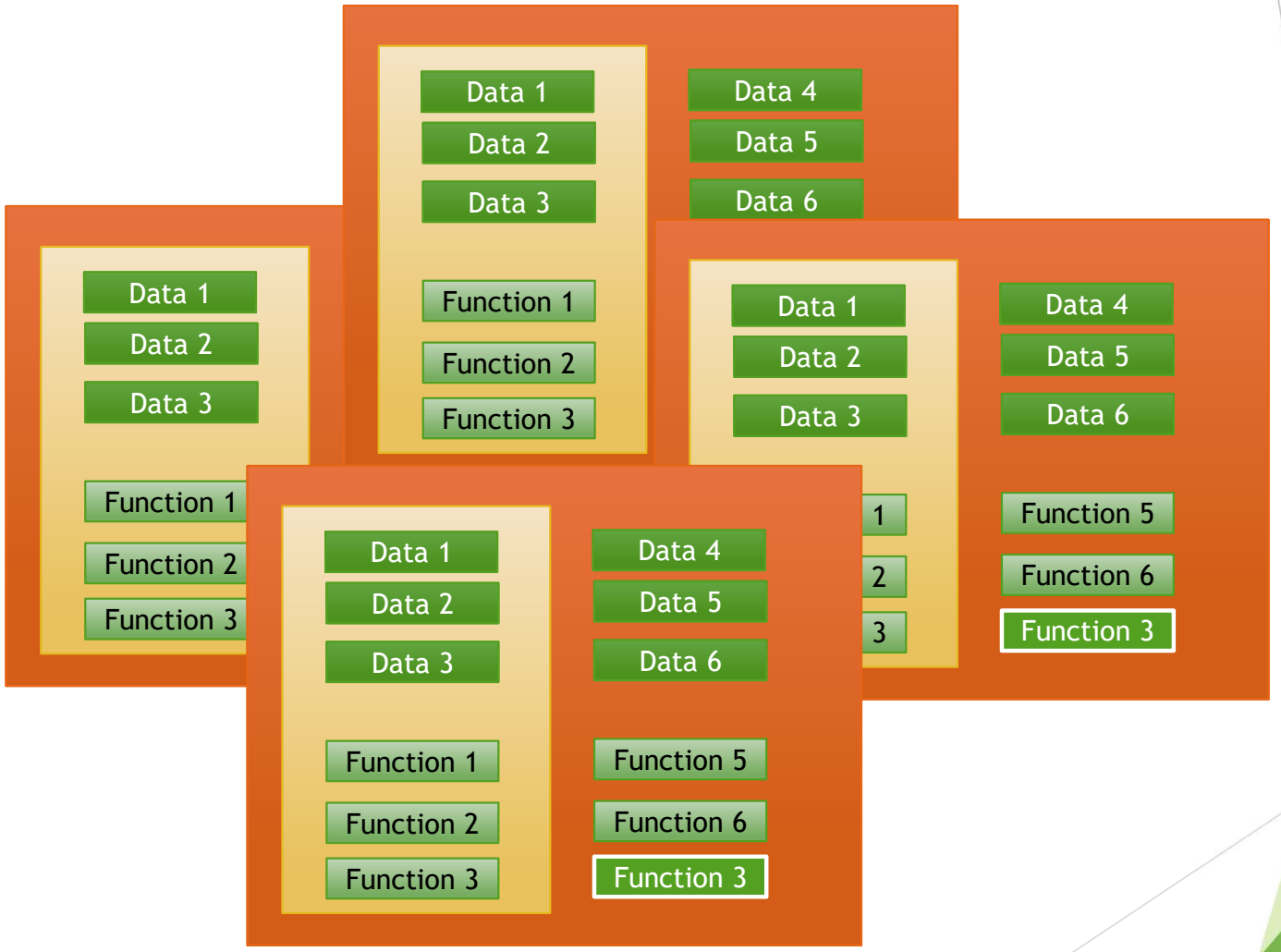
Plan -> class

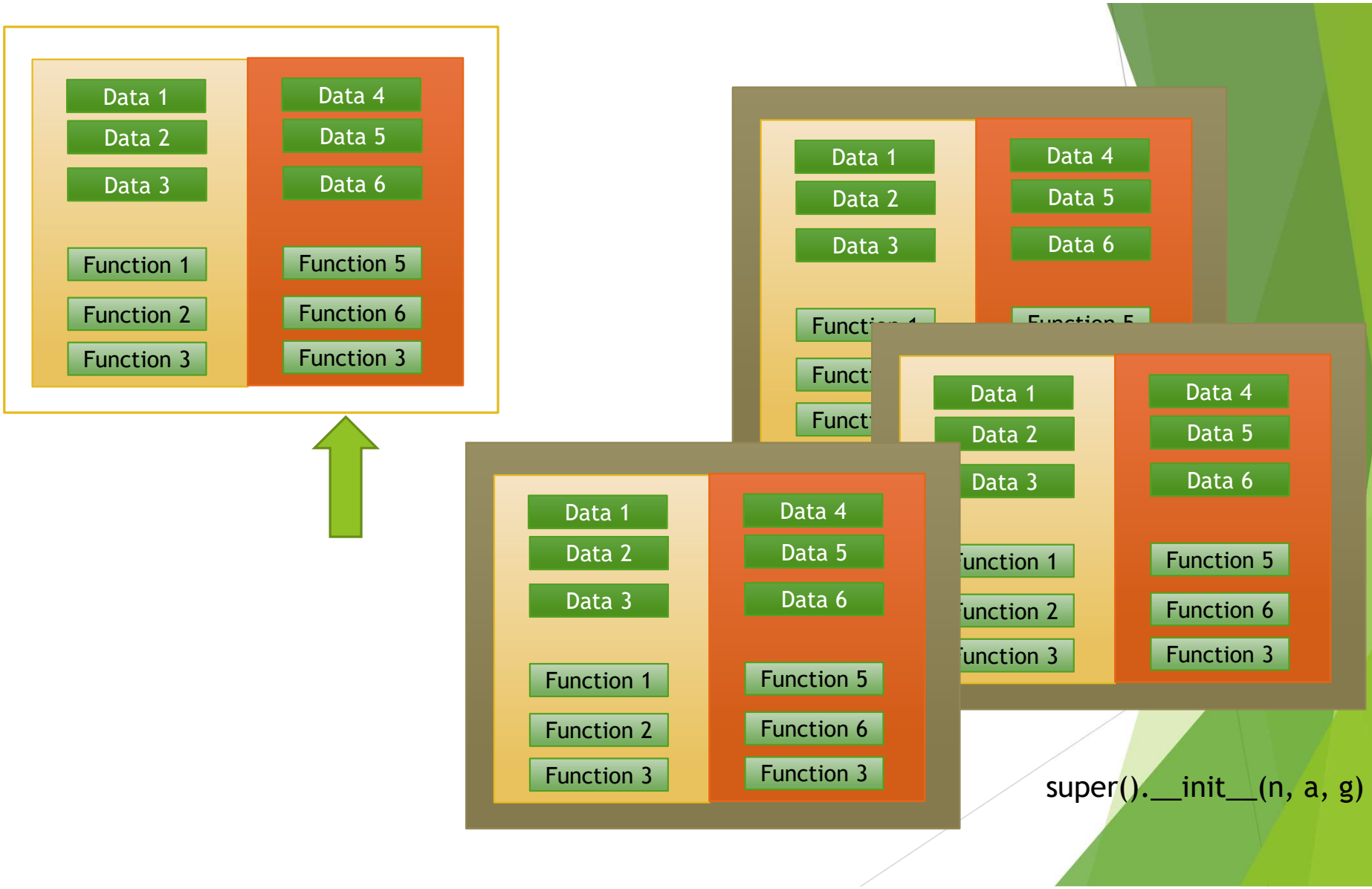
D

Implement plan -> creating objects


Data 1
Data 2
Data 3
Data 4
Data 5
Data 6
Function 1
Function 2
Function 3
Function 5
Function 6
Function 3
Data 1
Data 2
Data 3
Data 4
Data 5
Data 6
Function 1
Function 2
Function 3
Function 5
Function 6
Function 3

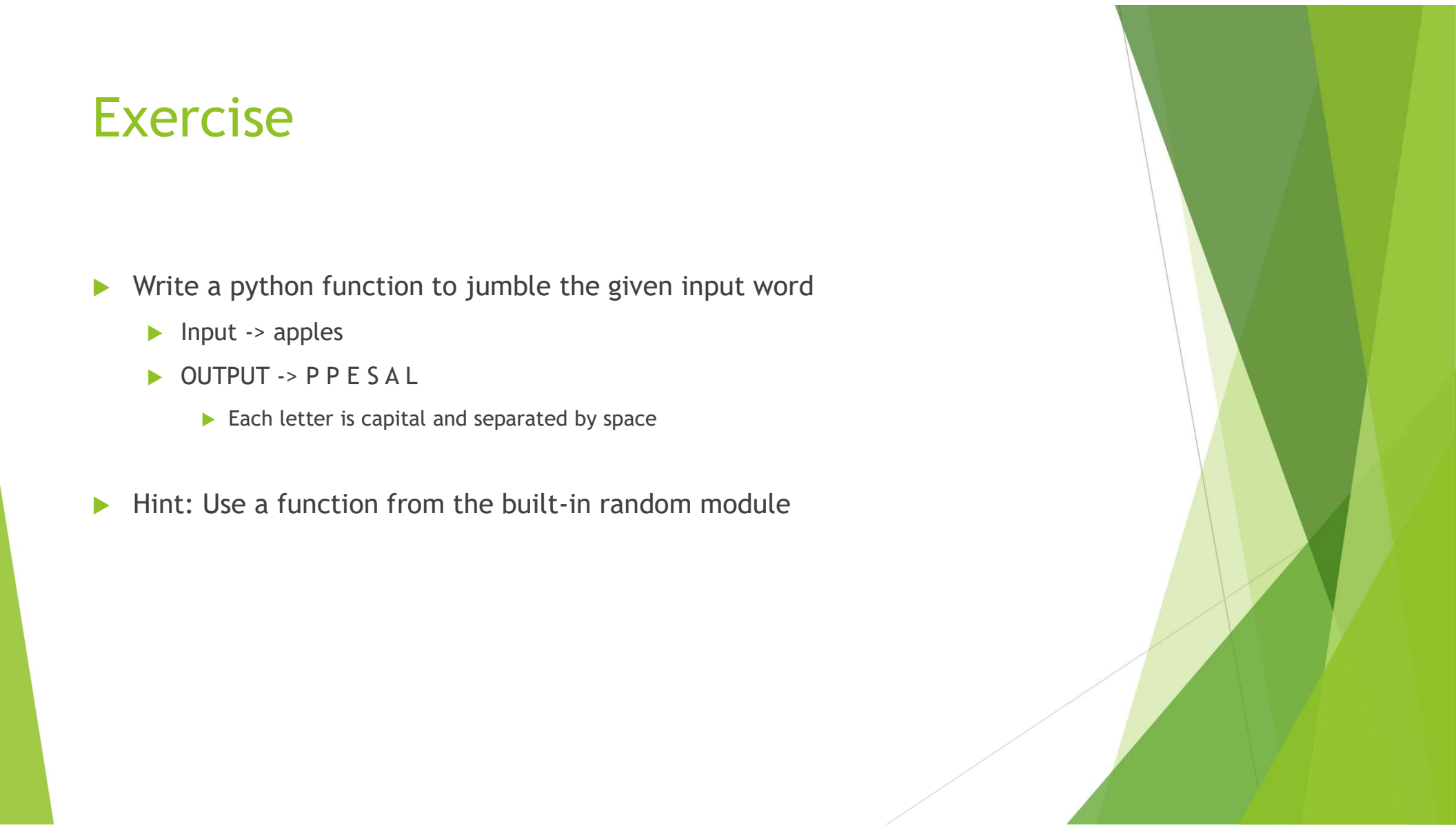
D


Data 1
Data 2
Data 3
Function 1
Function 2
Function 3
Data 4
Data 5
Data 6
Function 5
Function 6
Function 3


Data 1
Data 2
Data 3
Data 4
Data 5
Data 6
Function 1
Function 2
Function 3
Function 5
Function 6
Function 3
Data 1
Data 2
Data 3
Data 4
Data 5
Data 6
Data 1
Data 2
Data 3
Data 4
Data 5
Data 6
Function 1
Function 2
Function 3
Function 5
Function 6
Function 3
Data 1
Data 2
Data 3
Data 4
Data 5
Data 6
Function 1
Function 2
Function 3
Function 5
Function 6
Function 3
super().__init__(n, a, g)

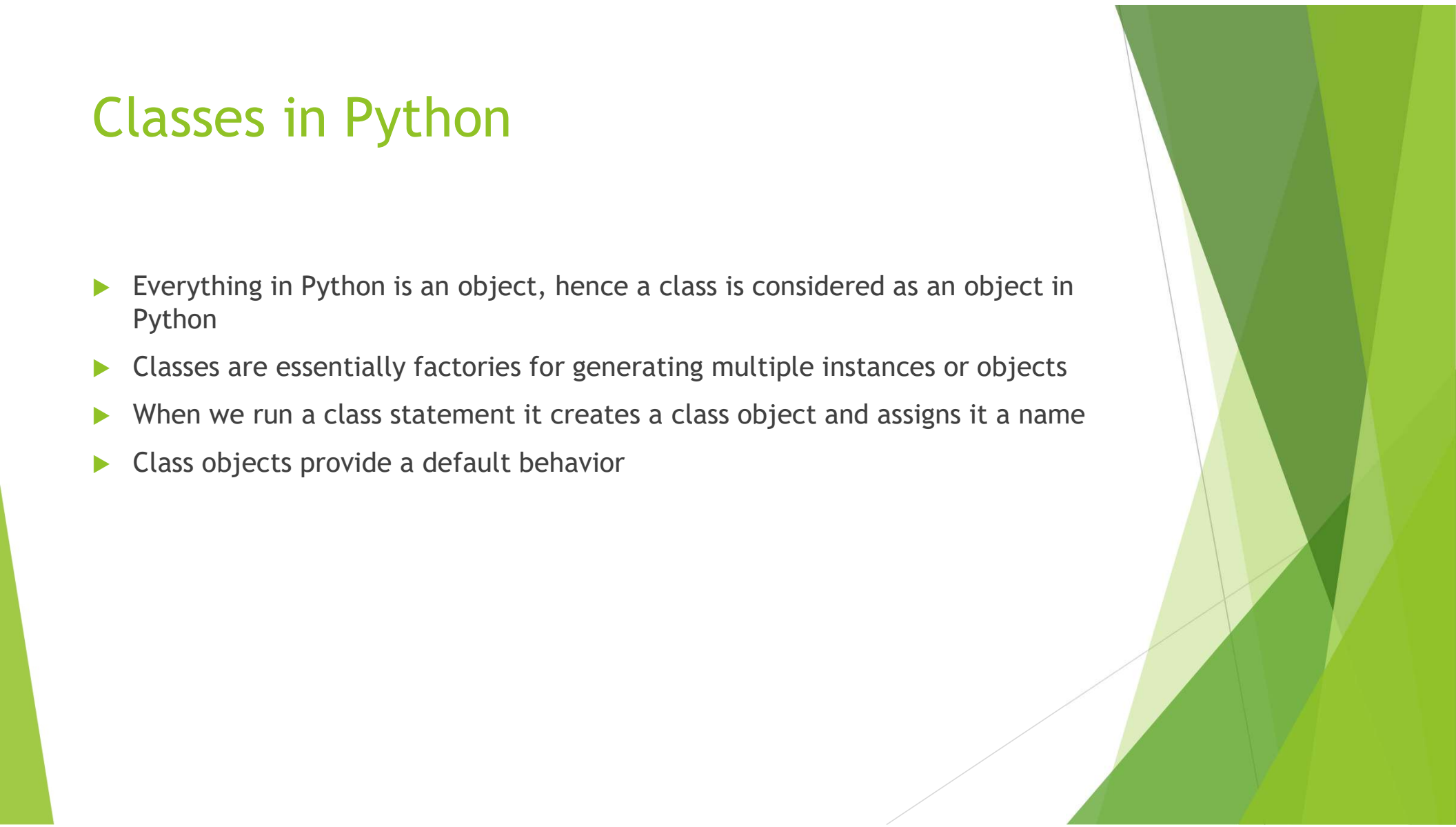
Exercise

- Write a python function to jumble the given input word
  - Input -> apples
  - OUTPUT -> P P E S A L
    - Each letter is capital and separated by space
- Hint: Use a function from the built-in random module



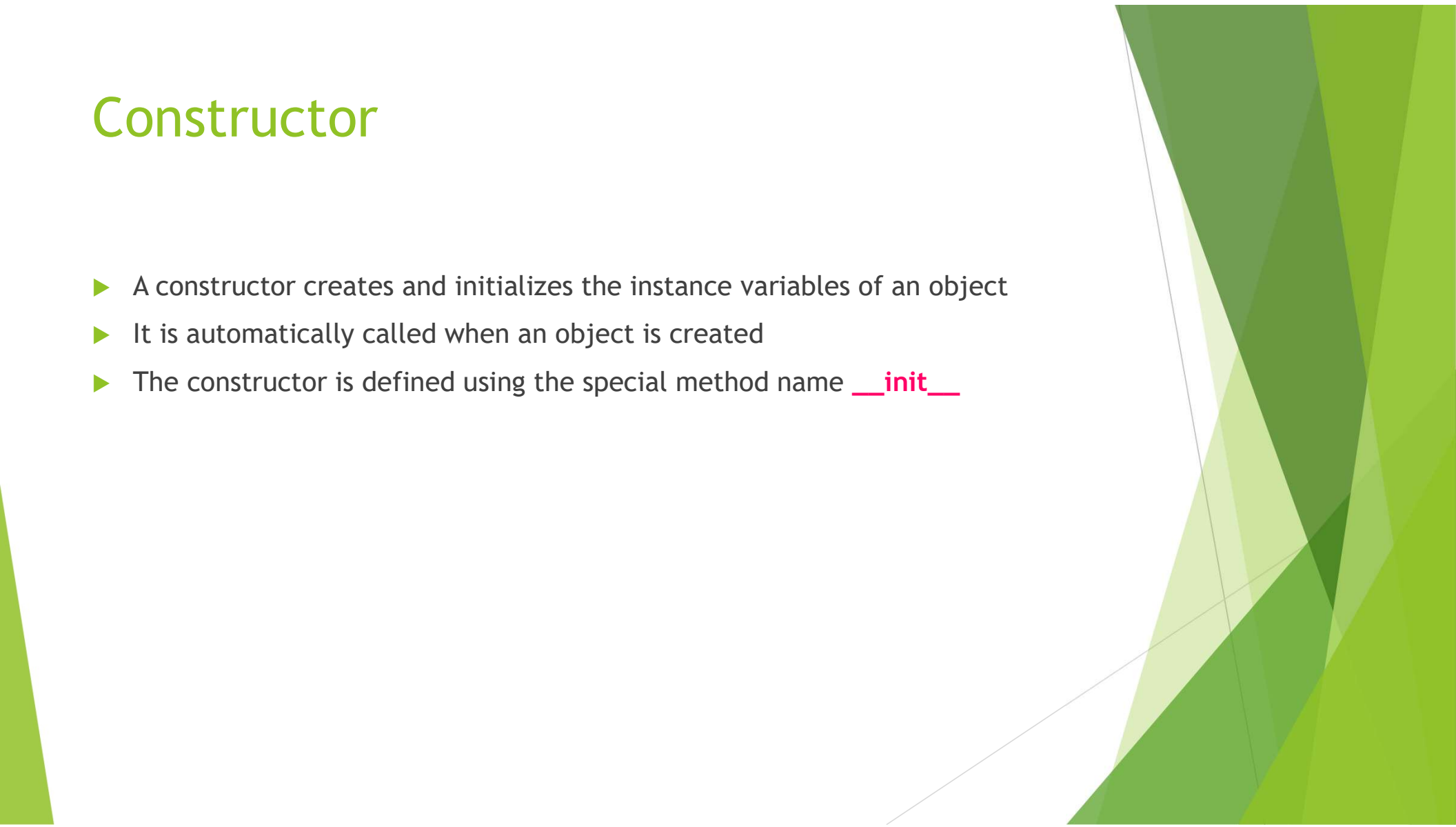
Classes in Python

- Everything in Python is an object, hence a class is considered as an object in Python
- Classes are essentially factories for generating multiple instances or objects
- When we run a class statement it creates a class object and assigns it a name
- Class objects provide a default behavior



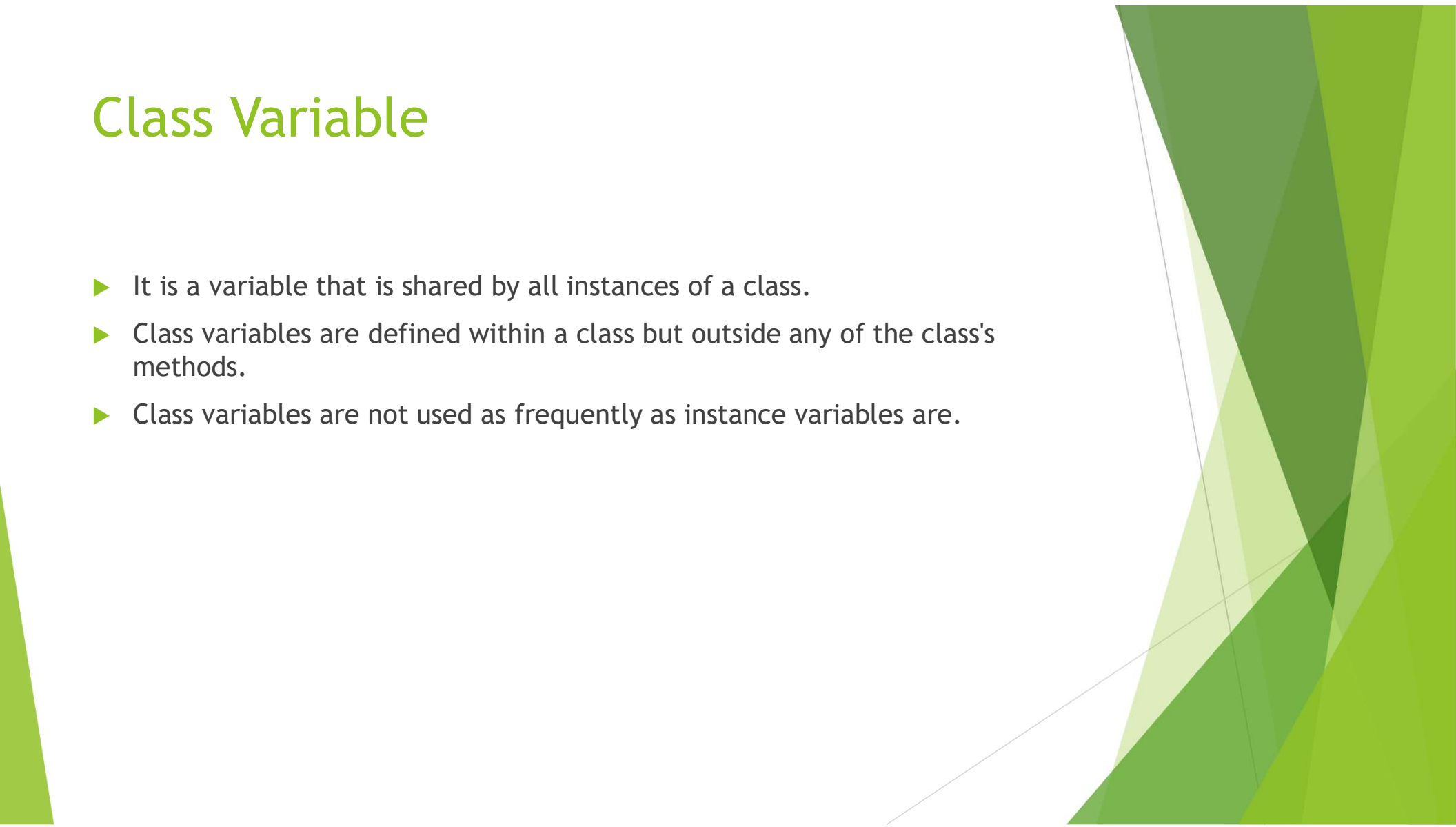
Constructor

- A constructor creates and initializes the instance variables of an object
- It is automatically called when an object is created
- The constructor is defined using the special method name **__init__**



Class Variable

- It is a variable that is shared by all instances of a class.
- Class variables are defined within a class but outside any of the class's methods.
- Class variables are not used as frequently as instance variables are.

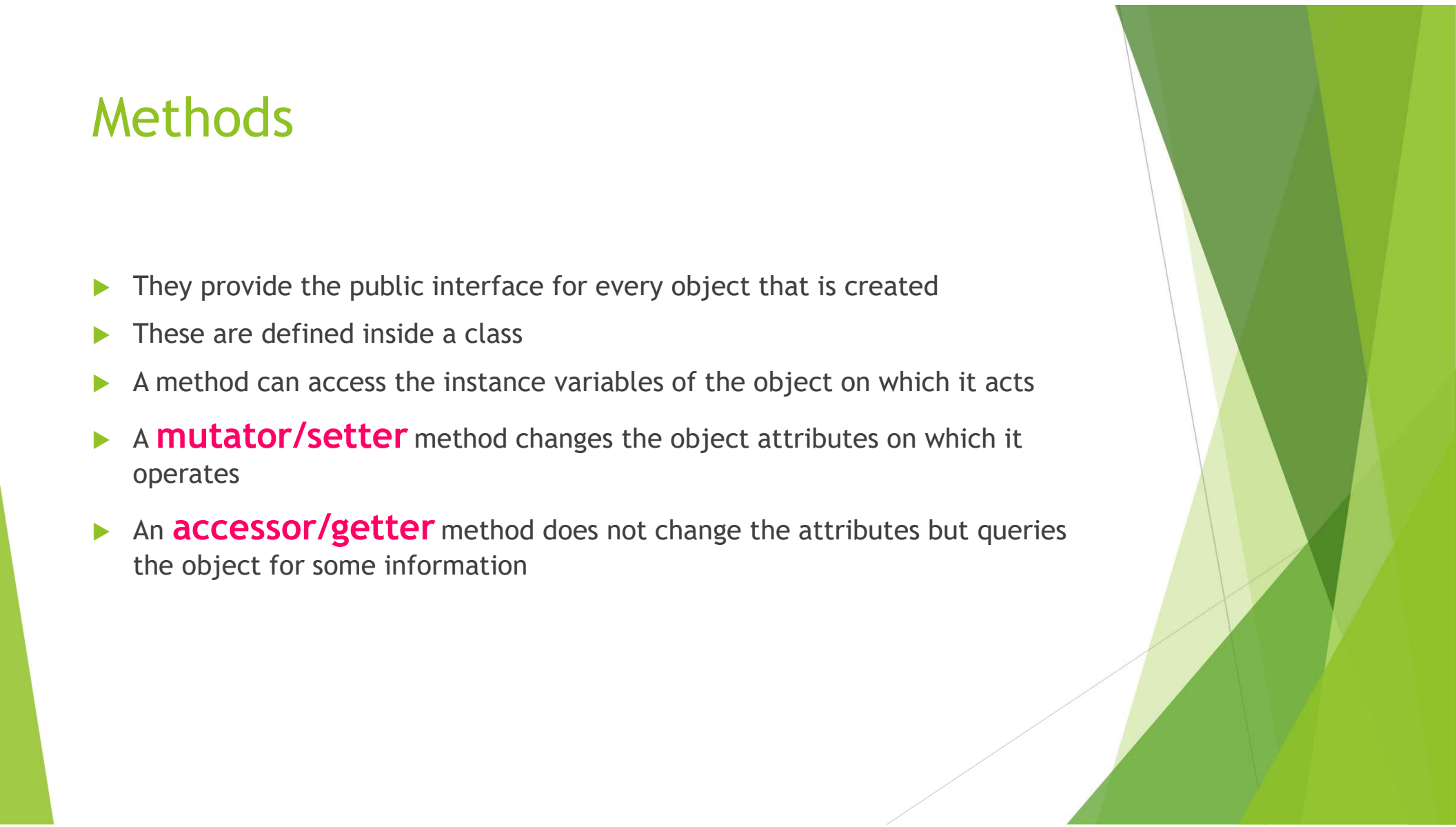


Instance Variable

- A variable that is defined inside a method and belongs only to the current instance of a class

Methods

- They provide the public interface for every object that is created
- These are defined inside a class
- A method can access the instance variables of the object on which it acts
- A **mutator/setter** method changes the object attributes on which it operates
- An **accessor/getter** method does not change the attributes but queries the object for some information

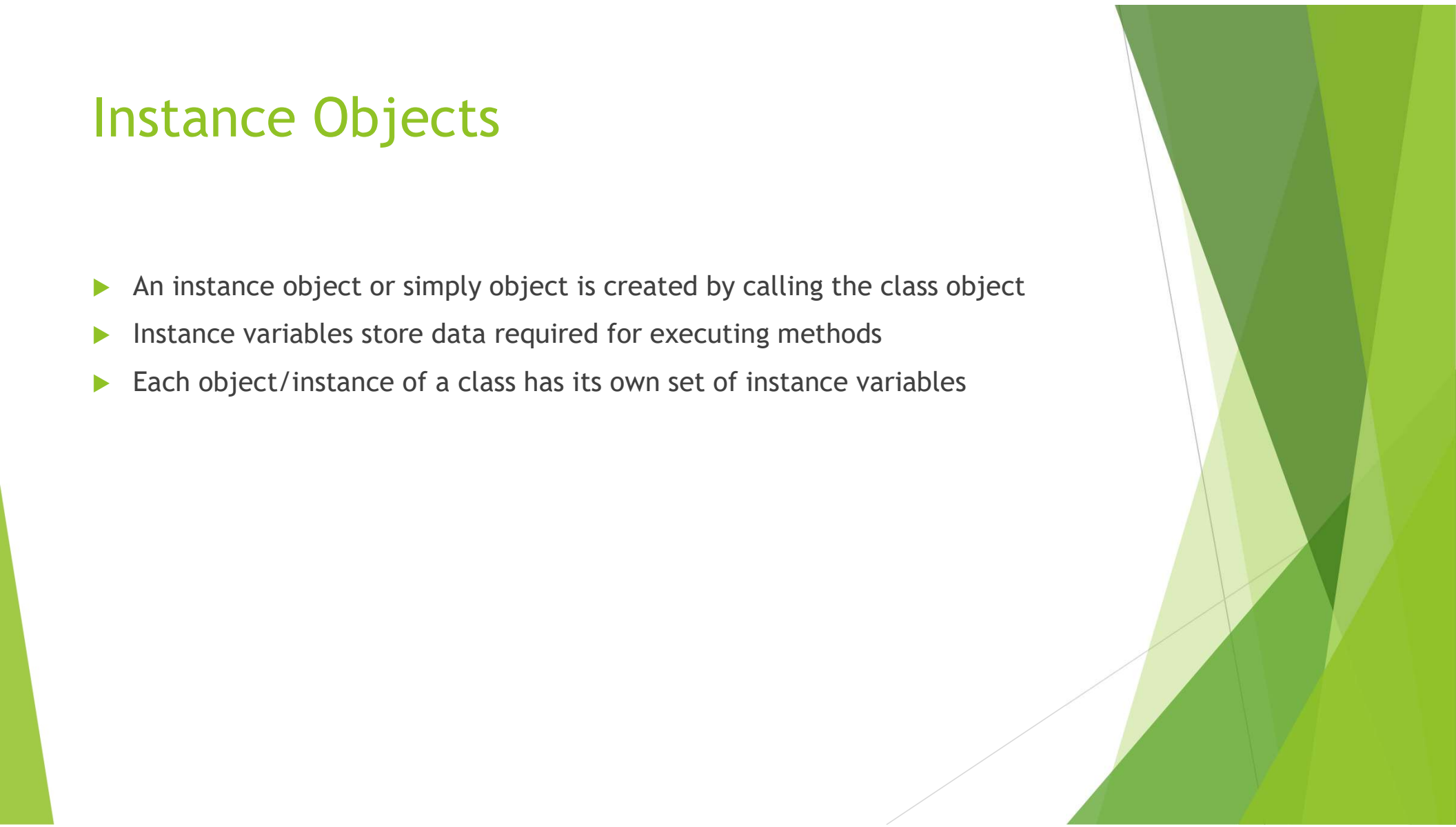


The **self** Argument

- You declare other class methods like normal functions with the exception that the first argument to each method is **self**.
- Python adds the **self** argument to the list for you; you do not need to include it when you call the methods.

Instance Objects

- An instance object or simply object is created by calling the class object
- Instance variables store data required for executing methods
- Each object/instance of a class has its own set of instance variables

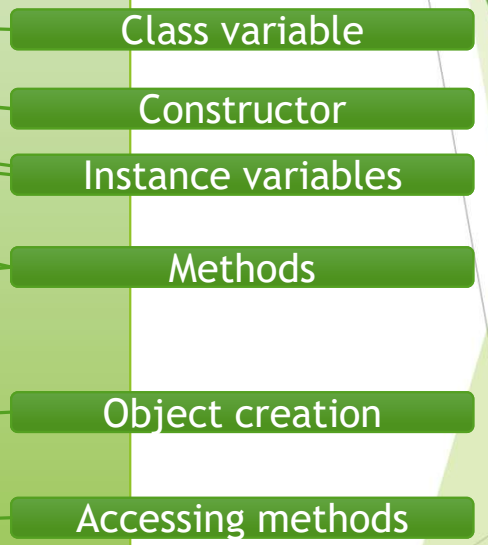


Accessing Methods

- You access the object's attributes using the dot operator with object

Example

```
class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print "Total Employee %d" % Employee.empCount

   def displayEmployee(self):
      print "Name : ", self.name,  ", Salary: ", self.salary


emp1 = Employee("Kumar", 2000)
emp2 = Employee("Abhinav", 5000)

emp1.displayEmployee()
emp2.displayEmployee()

print "Total Employee %d" % Employee.empCount
```

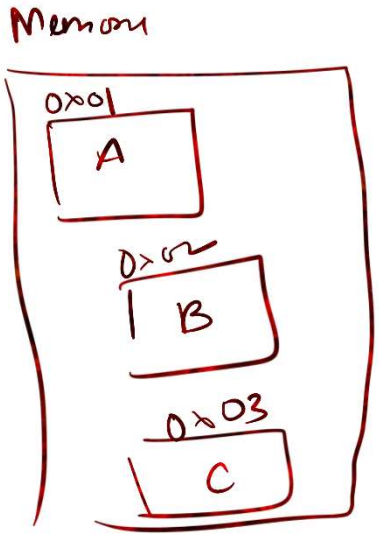


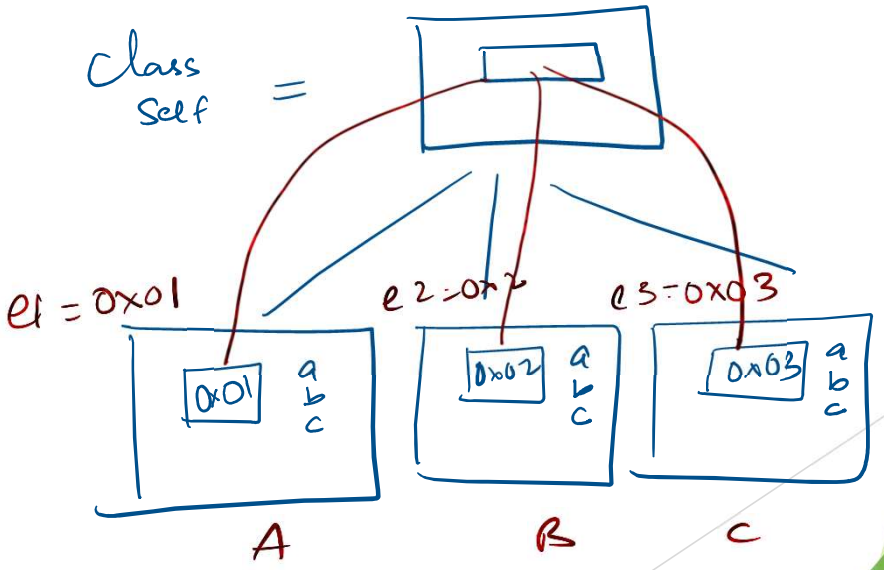
Special Functions to Access Attributes

- Instead of using the normal statements to access attributes, you can use the following functions –
  - **getattr(obj, name[, default])** : to access the attribute of object.
  - **hasattr(obj,name)** : to check if an attribute exists or not.
  - **setattr(obj,name,value)** : to set an attribute. If attribute does not exist, then it would be created.
  - **delattr(obj, name)** : to delete an attribute.

Example

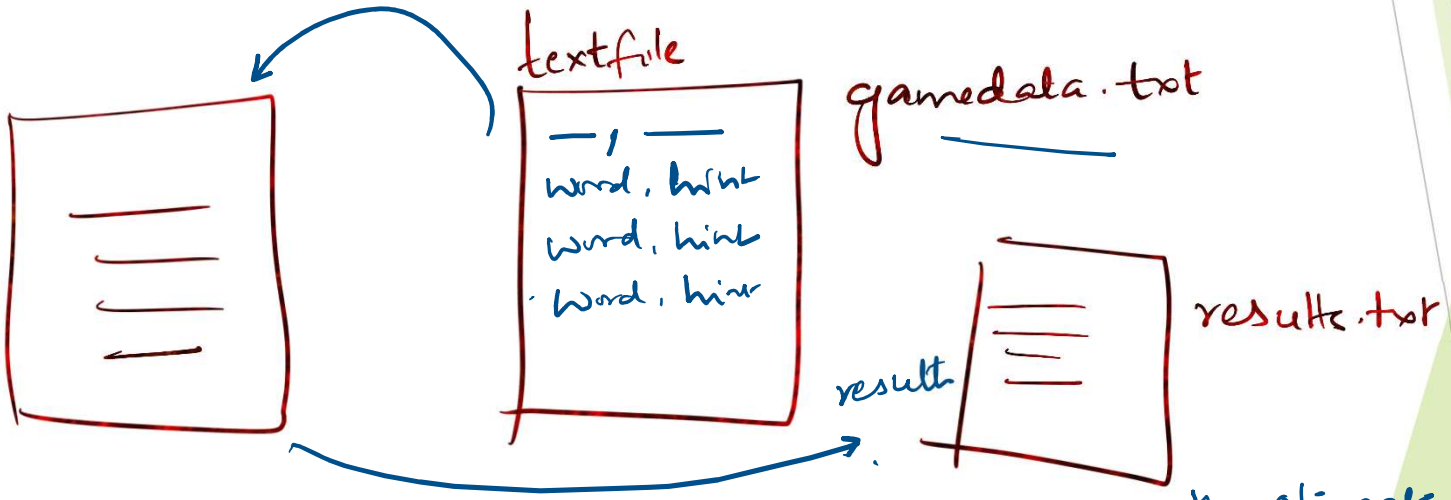
```
hasattr(emp1, 'age')    # Returns true if 'age' attribute exists
getattr(emp1, 'age')    # Returns value of 'age' attribute
setattr(emp1, 'age', 8) # Set attribute 'age' at 8
delattr(empl, 'age')    # Delete attribute 'age'
```





Exercise

OOP - Word Jumble game



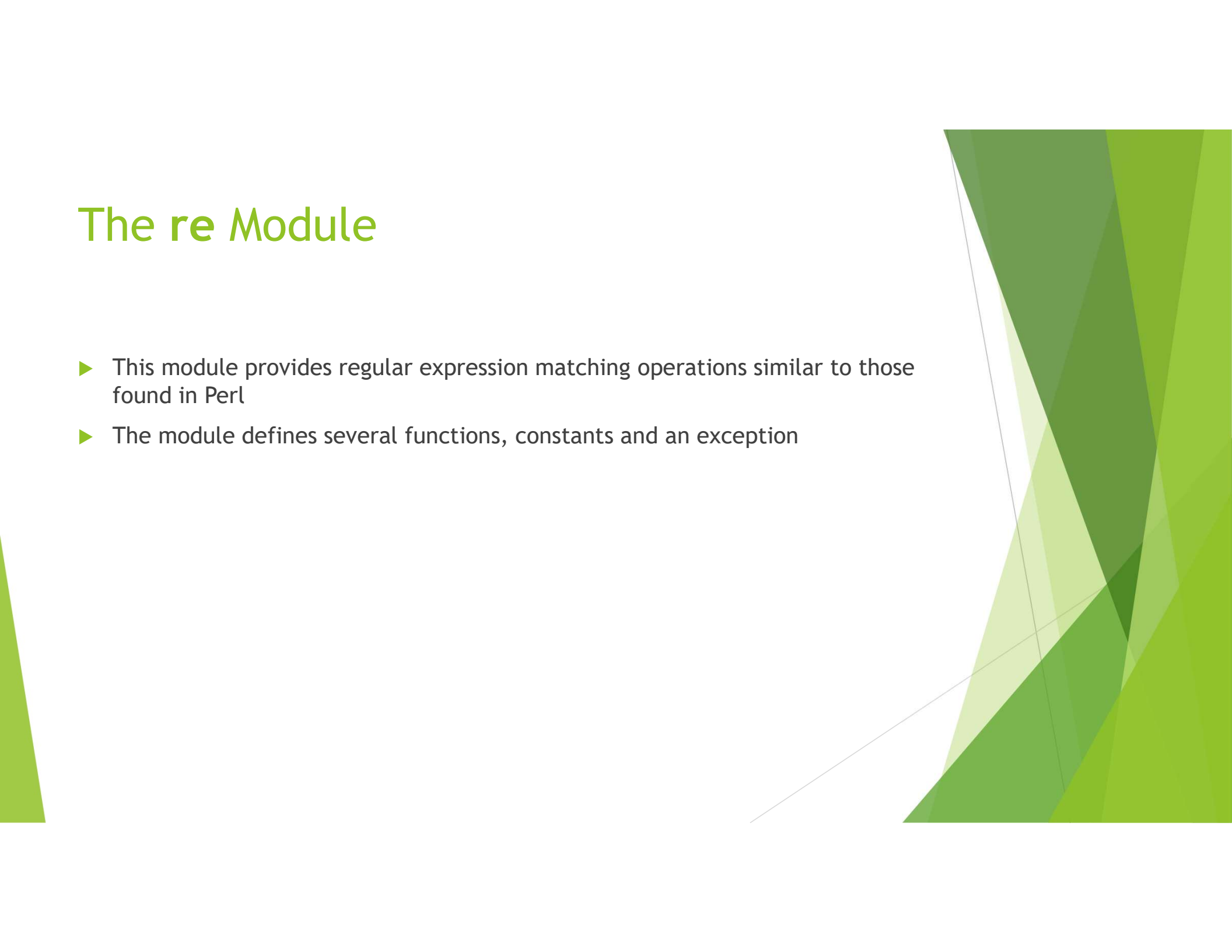
file should be an input to the object

p = WordJumbleGame('Anil', 'words12.txt')

Inheritance [run() Jumble()
results() → override]

The **re** Module

- This module provides regular expression matching operations similar to those found in Perl
- The module defines several functions, constants and an exception



re.match()

- Attempts to match RE pattern to string with optional flags; return match object on success, **None** on failure
- Syntax: **match(pattern, string, flags=0)**

```
>>> string = 'pamplemousse'
>>> import re
>>> f = re.match(r'pam', string)
>>> f
<_sre.SRE_Match object; span=(0, 3), match='pam'>
>>> type(f)
<class '_sre.SRE_Match'>
>>> f = re.match(r'mou', string)
>>> f
>>> type(f)
<class 'NoneType'>
```

re.search()

- Search for first occurrence of RE pattern within string with optional flags; return match object on success, **None** on failure
- Syntax: **search(pattern, string, flags=0)**

```
>>> string = 'Ratatouille'
>>> f = re.match(r'tou', string)
>>> f
>>> f = re.search(r'tou', string)
>>> f
<_sre.SRE_Match object; span=(4, 7), match='tou'>
```

re.findall()

- Look for all (non-overlapping) occurrences of pattern in string; return a list of matches
- Syntax: **findall(pattern, string[,flags])**

```
>>> string = 'if stu chews shoes, should stu choose the shoes he chews'
>>> f = re.findall(r'stu', string)
>>> f
['stu', 'stu']
>>> h = re.findall(r'ho', string)
>>> h
['ho', 'ho', 'ho', 'ho']
>>> print(re.findall(r'\b[a-z]ho*', string))
['ch', 'sho', 'sho', 'choo', 'th', 'sho', 'ch']
>>> print(re.findall(r'\b[a-z]ho\w*', string))
['shoes', 'should', 'choose', 'shoes']
```

re.finditer()

- This is same as **findall()** except returns an iterator instead of a list; for each match, the iterator returns a match object

```
>>> string = 'if stu chews shoes, should stu choose the shoes he chews'
>>> g = re.finditer(r'stu', string)
>>> g
<callable_iterator object at 0x016ABE50>
>>>
>>> for i in (re.finditer(r'\b[a-z]ho\w*', string)):
     print ("'{g}' was found between the indices {s}".format(g=i.group(), s=i.span()))


'shoes' was found between the indices (13, 18)
'should' was found between the indices (20, 26)
'choose' was found between the indices (31, 37)
'shoes' was found between the indices (42, 47)
```

re.sub()/re.subn()

- Syntax: **sub(pattern, replacement, string, max=0)**
- Replace all occurrences of the RE pattern in string with replacement, substituting all occurrences unless max provided
- **subn()** is same as **sub()** but in addition, it returns the number of substitutions made)

```
>>> string = 'twikle twikle little star'
>>> re.sub('twikle','twinkle',string)
'twinkle twinkle little star'
>>> string
'twikle twikle little star'
>>> re.subn('twikle','twinkle',string)
('twinkle twinkle little star', 2)
```

start()/end()

- Return the indices of the start and end of the substring matched by group; group defaults to zero (meaning the whole matched substring).
- Return -1 if group exists but did not contribute to the match.

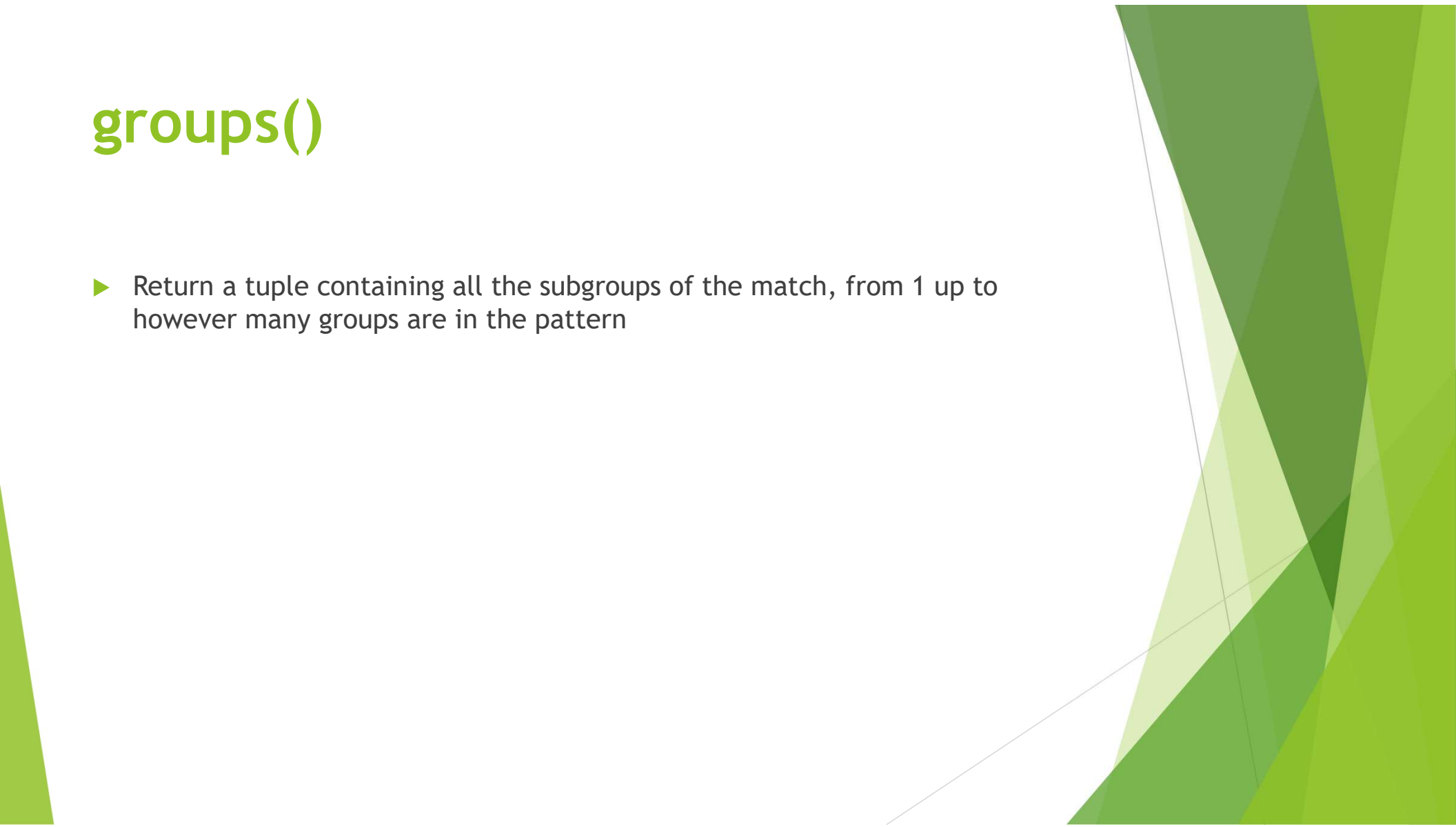
```
>>> email = "tony@tiremove_thisger.net"
>>> m = re.search("remove_this", email)
>>> email[:m.start()] + email[m.end():]
'tony@tiger.net'
```

group()

- Returns one or more subgroups of the match.
  - If there is a single argument, the result is a single string
  - If there are multiple arguments, the result is a tuple with one item per argument.
  - Without arguments, *group1* defaults to zero (the whole match is returned).
  - If a *groupN* argument is zero, the corresponding return value is the entire matching string
  - If it is in the inclusive range [1..99], it is the string matching the corresponding parenthesized group

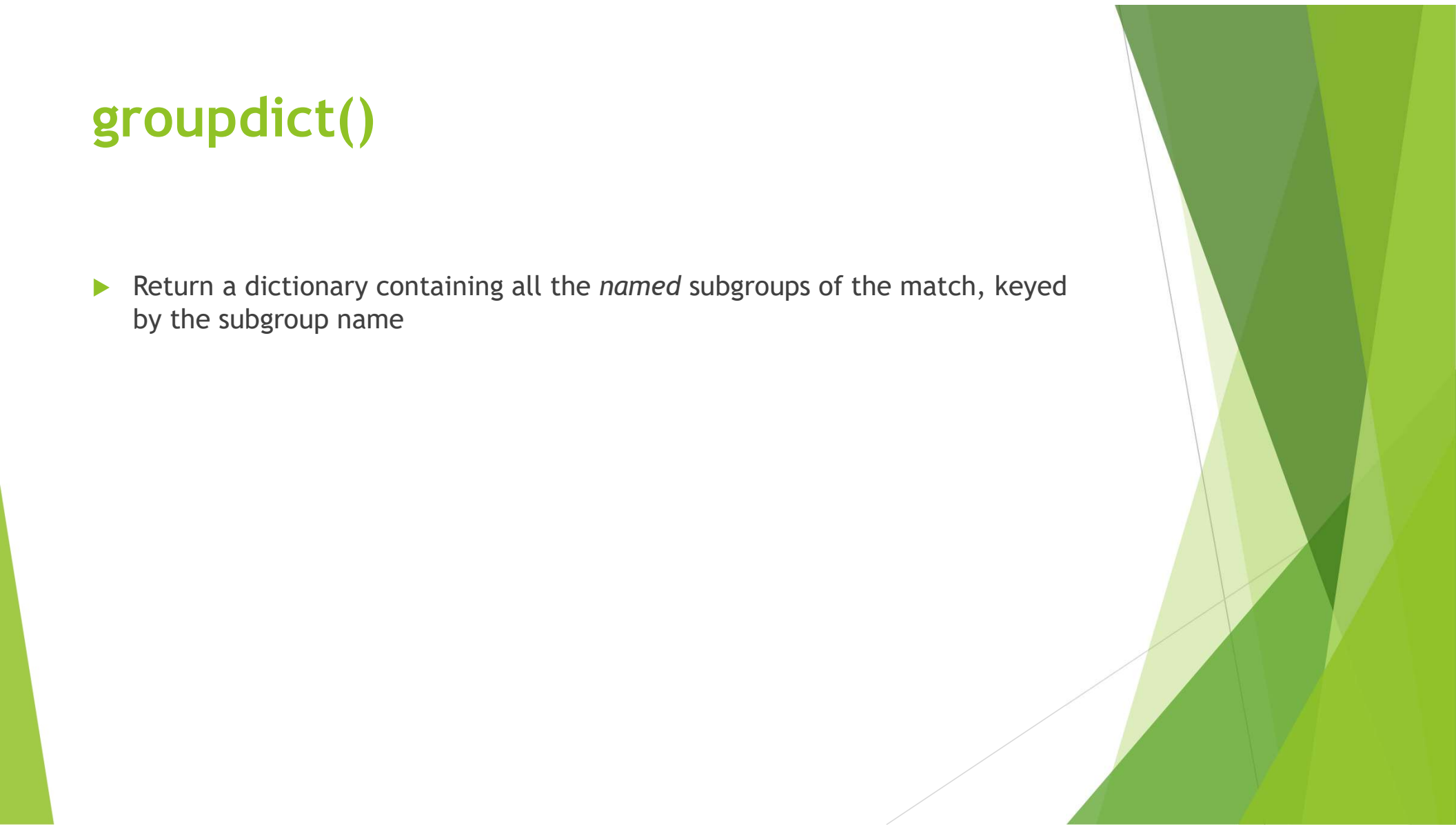
groups()

- Return a tuple containing all the subgroups of the match, from 1 up to however many groups are in the pattern



groupdict()

- Return a dictionary containing all the *named* subgroups of the match, keyed by the subgroup name



Grouping Examples

```
>>> m = re.match(r"(\d+)\.(\d+)", "24.1632")
>>> m.group()
'24.1632'
>>> m.group(1)
'24'
>>> m.group(2)
'1632'
>>> m = re.match(r"\d+\.\d+", "24.1632")
>>> m.group()
'24.1632'
>>> m.group(1)
Traceback (most recent call last):
  File "<pyshell#60>", line 1, in <module>
    m.group(1)
IndexError: no such group
```

```
>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
>>> m.groups()
('Isaac', 'Newton')
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
>>> m.groups()
('Malcolm', 'Reynolds')
>>> m.groupdict()
{'first_name': 'Malcolm', 'last_name': 'Reynolds'}
```

Metacharacters

Character	Description
.	The dot stands for any character (letter, digit, or special character) except \n.
[…]	Characters within square brackets build a character class. Any character found in the class returns a true value for the expression. A - indicates a range of values.
[^..]	Any character that is not in the list is a match. This negates the character class.
\	This causes the character following \ to be taken literally. There are many characters with special meanings in regular expressions (for example, ., *, or +). To match a + sign, prefix it with a backslash.
|	The or operator causes multiple patterns to be matched alternatively.

Metacharacters: Examples

```
# regex_experiments.py

import re
data = ['ab', 'abc', 'a5e', 'a6f', '123 a6c', 'a5b', 'a55b', 'a555b', 'a5555b',
        'a55555b', 'a555555b', 'a5xb', '1/4', '3+2=5', 'def ghi', 'abc ab']
for item in data:
     m = re.match(r'a.c', item)
     if m:
          print m.group() + ' matched in ' + '\'' + item + '\''
```

```
abc matched in 'abc'
abc matched in 'abc ab'
```

```
# regex_experiments.py

import re
data = ['ab', 'abc', 'a5e', 'a6f', '123 a6c', 'a5b', 'a55b', 'a555b', 'a5555b',
        'a55555b', 'a555555b', 'a5xb', '1/4', '3+2=5', 'def ghi', 'abc ab']
for item in data:
     m = re.search(r'a.c', item)
     if m:
          print m.group() + ' matched in ' + '\'' + item + '\''
```

```
abc matched in 'abc'
a6c matched in '123 a6c'
abc matched in 'abc ab'
```

Metacharacters: Examples

```
re.match(r'a.[abd-z]', item)
```

```
a5e matched in 'a5e'
a6f matched in 'a6f'
a5b matched in 'a5b'
a5x matched in 'a5xb'
```

```
re.search(r'a[0-9][a-z]', item)
```

```
a5e matched in 'a5e'
a6f matched in 'a6f'
a6c matched in '123 a6c'
a5b matched in 'a5b'
a5x matched in 'a5xb'
```

```
re.search(r'a[^0-9][^0-9]', item)
```

```
abc matched in 'abc'
abc matched in 'abc ab'
```

Metacharacters: Examples

```
re.search(r'[0-9]\+[0-9]', item)
```

```
3+2 matched in '3+2=5'
```

```
re.search(r'bc|a6', item)
```

```
bc matched in 'abc'
a6 matched in 'a6f'
a6 matched in '123 a6c'
bc matched in 'abc ab'
```

Default Character Class

- Default character class is a special predefined characters as shorthand for these character classes. The default characters are shown in the following table:

Predefined Character	Character Class	Negated Character	Negated Class
\d (digit)	[0-9]	**\D**	[^0-9]
\w (word-building char)	[a-zA-Z0-9_]	**\W**	[^a-zA-Z0-9_]
\s (white space)	[\r\t\n\f]	**\S**	[^\r\t\n\f]

Default Character Class

```
re.search(r'a\D\D', item)
```

```
abc matched in 'abc'
abc matched in 'abc ab'
```

```
re.search(r'...\s.', item)
```

```
123 a matched in '123 a6c'
def g matched in 'def ghi'
abc a matched in 'abc ab'
```

Anchors

- Anchors determine the edges of the search patterns. Patterns may be anchored to the start or end of strings or words as well as lines. The anchor characters are shown in the following table:

Anchor	Description
^	The beginning of the line
$	The end of the line
\b	A word boundary. To delimit a word, put `\b` in the front and at the end of the pattern. A word is everything that consists of `\w` characters that ends before a `\W` character or `newline`.
\B	This is the opposite of `\b`, which specifies that the word does not end at this point.

Anchors: Examples

```
re.search(r'a6', item)
```

```
a6 matched in 'a6f'
a6 matched in '123 a6c'
```

```
re.search(r'^a6', item)
```

```
a6 matched in 'a6f'
```

```
re.search(r'..c', item)
```

```
abc matched in 'abc'
a6c matched in '123 a6c'
abc matched in 'abc ab'
```

```
re.search(r'..c$', item)
```

```
abc matched in 'abc'
a6c matched in '123 a6c'
```

Anchors: Examples

```
re.search(r'ab\b', item)
```

```
ab matched in 'ab'
ab matched in 'abc ab'
```

```
re.search(r'ab\B', item)
```

```
ab matched in 'abc'
ab matched in 'abc ab'
```

Quantifiers

- To specify that a placeholder is repeated a number of times, a quantifier is used.

Quantifier	Description
`*`	The previous character is repeated zero or more times.
`+`	The previous character is repeated one or more times.
`?`	The previous character must appear exactly one time or not at all.
`{n}`	The previous character appears exactly n times.
`{m,n}`	The previous character appears from m to n times.
`{m,} {,n}`	The previous character appears m or more times.
`(..)`	This groups characters together for use in alternation.

Quantifiers: Examples

```
re.search(r'a5555*', item)
```

```
a555 matched in 'a555b'
a5555 matched in 'a5555b'
a55555 matched in 'a55555b'
a555555 matched in 'a555555b'
```

```
re.search(r'a5555+', item)
```

```
a5555 matched in 'a5555b'
a55555 matched in 'a55555b'
a555555 matched in 'a555555b'
```

```
re.search(r'a5?\D', item)
```

```
ab matched in 'ab'
ab matched in 'abc'
a5e matched in 'a5e'
a5b matched in 'a5b'
a5x matched in 'a5xb'
ab matched in 'abc ab'
```

Quantifiers: Examples

```
re.search(r'a5{4}\D', item)
```

```
a5555b matched in 'a5555b'
```

```
re.search(r'a5{3,5}\D', item)
```

```
a555b matched in 'a555b'
a5555b matched in 'a5555b'
a55555b matched in 'a55555b'
```

```
re.search(r'a5{4,}\D', item)
```

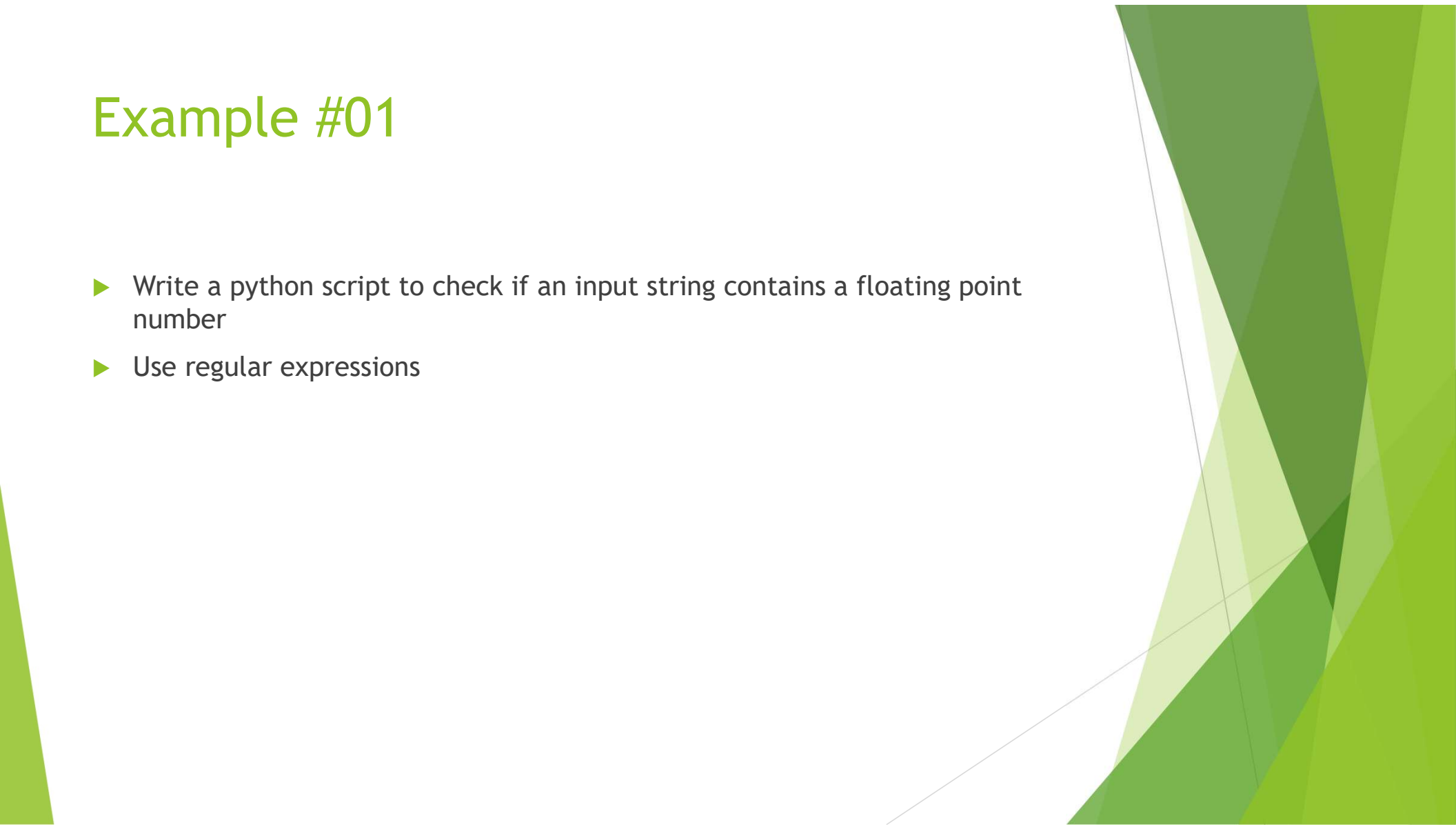
```
a5555b matched in 'a5555b'
a55555b matched in 'a55555b'
a555555b matched in 'a555555b'
```

```
re.search(r'(55){2}', item)
```

```
5555 matched in 'a5555b'
5555 matched in 'a55555b'
5555 matched in 'a555555b'
```

Example #01

- Write a python script to check if an input string contains a floating point number
- Use regular expressions



Solution

```
import re

expr = '37.0 degree centigrade is equal to +98.6 farhenheit'

pattern = r'-|\+?\d+\.?\d*|\.?\d+'

print expr
print 'Trying to find a floating point numbers in the statement...'

match = re.findall(pattern, expr)
if match:
    print 'Following numbers were found', match
else:
    print 'Seems like there is no floating point number'
```

```
======= RESTART: E:/Python27/mindful_examples/regex/floating_point.py =======
37.0 degree centigrade is equal to +98.6 farhenheit
Trying to find a floating point numbers in the statement...
Following numbers were found ['37.0', '+98.6']
```

Explain the results of the following patterns in the above code:

```
pattern = r'(-|\+)?(\d+\.?\d*|\.?\d+)‘
pattern = r'(-|\+)?(\d+\.?\d*)|(\.?\d+)'
```