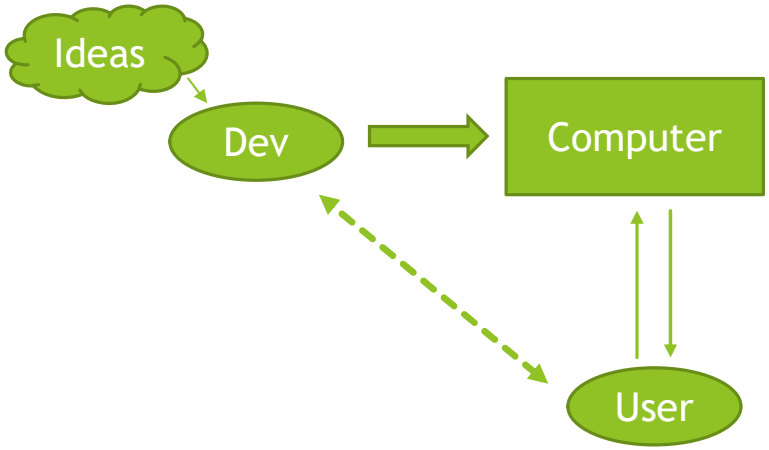
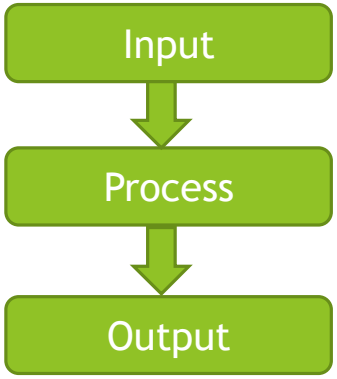
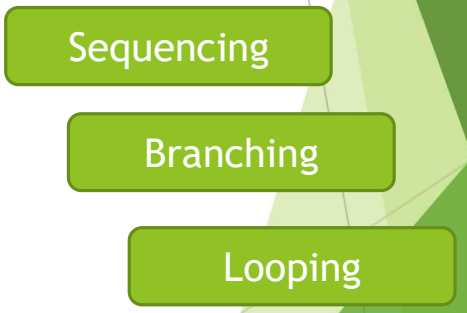
Problem Solving with Python

Computer Programming Concepts Review

- Computer is a hardware capable of performing certain task
- It needs to be instructed on what task to perform, hence the use of computer languages

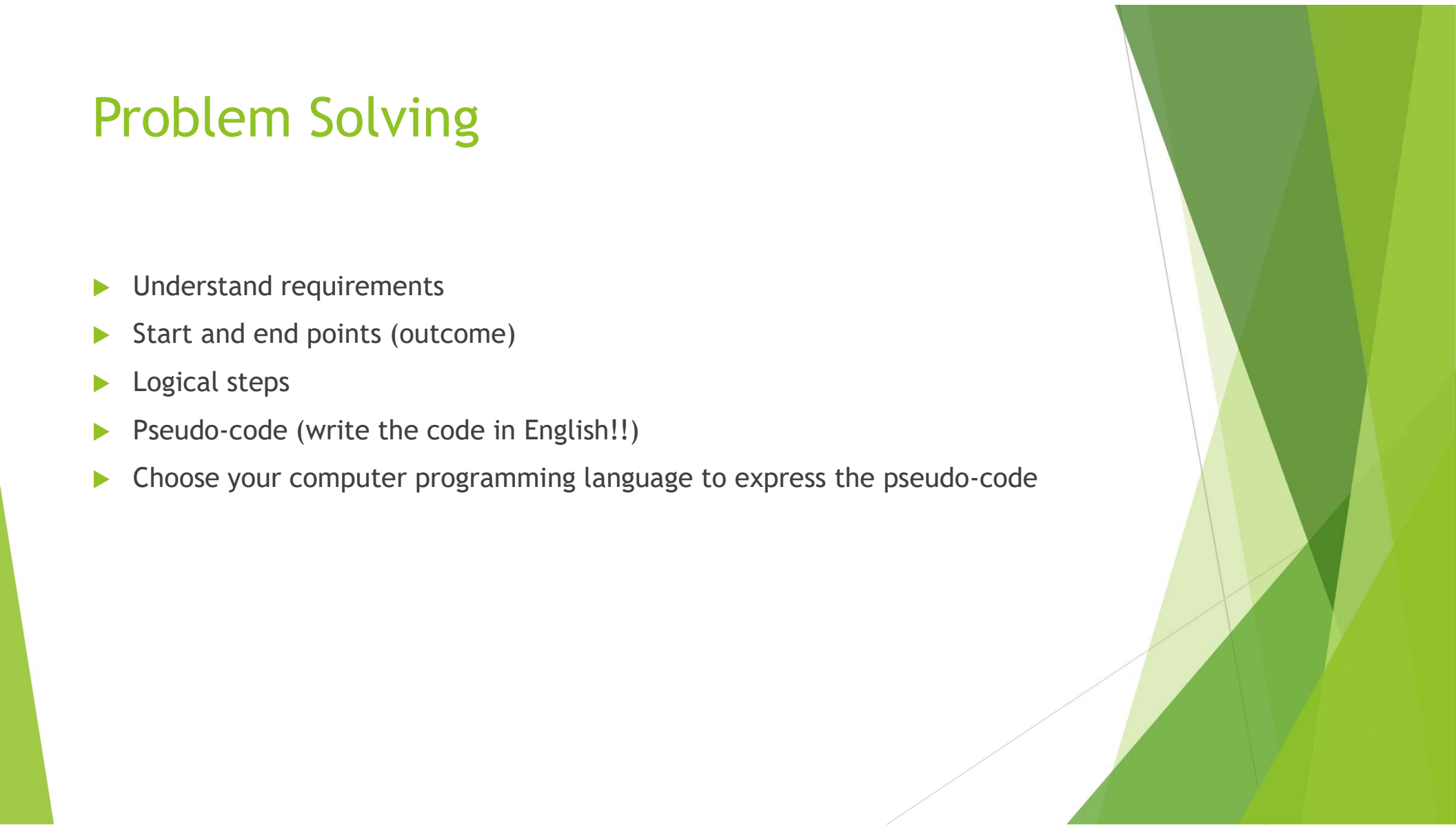






Problem Solving

- Understand requirements
- Start and end points (outcome)
- Logical steps
- Pseudo-code (write the code in English!!)
- Choose your computer programming language to express the pseudo-code

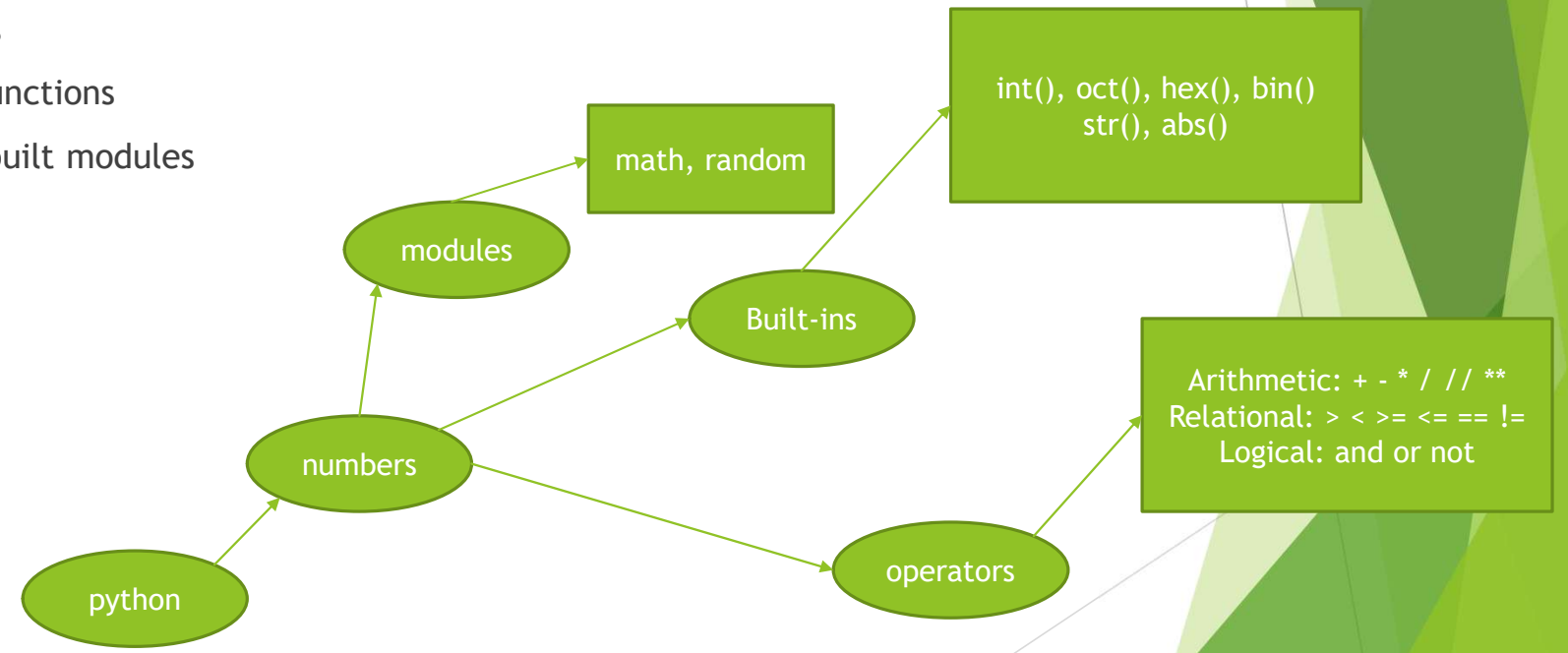


Python Installation

https://www.python.org/downloads/windows/

Numbers

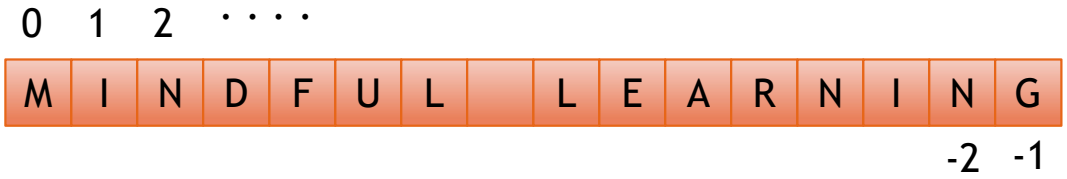
- All numeric values
- Python provides the following for treating numbers
  - Operators
  - In-built functions
  - Some in-built modules



Strings

- Collection of characters considered as a single entity, immutable by nature
- Structure

S = ‘MINDFUL LEARNING’



- Access characters using subscripting
  - **[start:end:interval]**
- Treatment
  - Operators
  - In-built functions

Lists

- Ordered collection of objects, mutable by nature
- Structure and declaration:

```
L = ['C', 'C++', 'Perl', 'Python', 'Java']
```

- Each element is indexed
- Elements can be replaced (mutable)
- Access characters using subscripting
  - **[start:end:interval]**
- Treatment
  - Operators
  - In-built functions

Tuples

- Ordered collection of objects, immutable
- Structure and declaration:

```
T = ('C', 'C++', 'Perl', 'Python', 'Java')
```

- Each element is indexed
- Elements cannot be replaced (immutable)
- Access characters using subscripting
  - **[start:end:interval]**
- Treatment
  - Operators
  - In-built functions (no many since tuples are immutable structures)

Sets

- Unique collection of objects, unordered, mutable
- Structure and declaration:

```
S = {'C', 'C++', 'Perl', 'Python', 'Java'}
```

- No indexes
- Elements can be replaced (mutable)
- Cannot access characters using subscripting (unordered)
- Treatment
  - Operators (supports operations according to mathematical set theory)
  - In-built functions

Dictionaries

- Unordered collections, key-values pairs, mutable, unique
- Structure and declaration:

```
D = {'Name' : 'Mark', 'EID' : 123456}
```

- Values access using respective keys
- Elements can be replaced (mutable)
- Treatment
  - Operators
  - In-built functions

Branching

- Conditional branching is achieved using the if..elif..else construct
- Syntax:

```
if <condition> :
    <statements>
elif <condition> :
    <statements>
.
.
.
else:
    <statements>
```

Looping using **for**

- The for construct is used to repeat a specific set of statements for a given number of times
- Syntax:

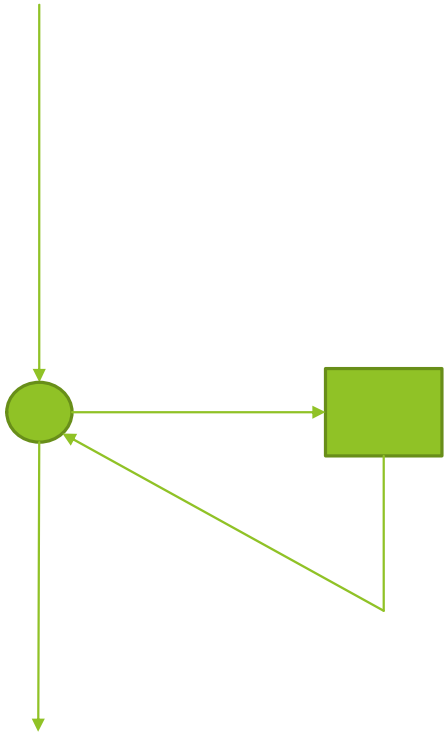
```
for <var> in <iterator>:
    <statements>
```

Looping using **while**

- The for construct is used to repeat a specific set of statements until a given condition fails
- Syntax:

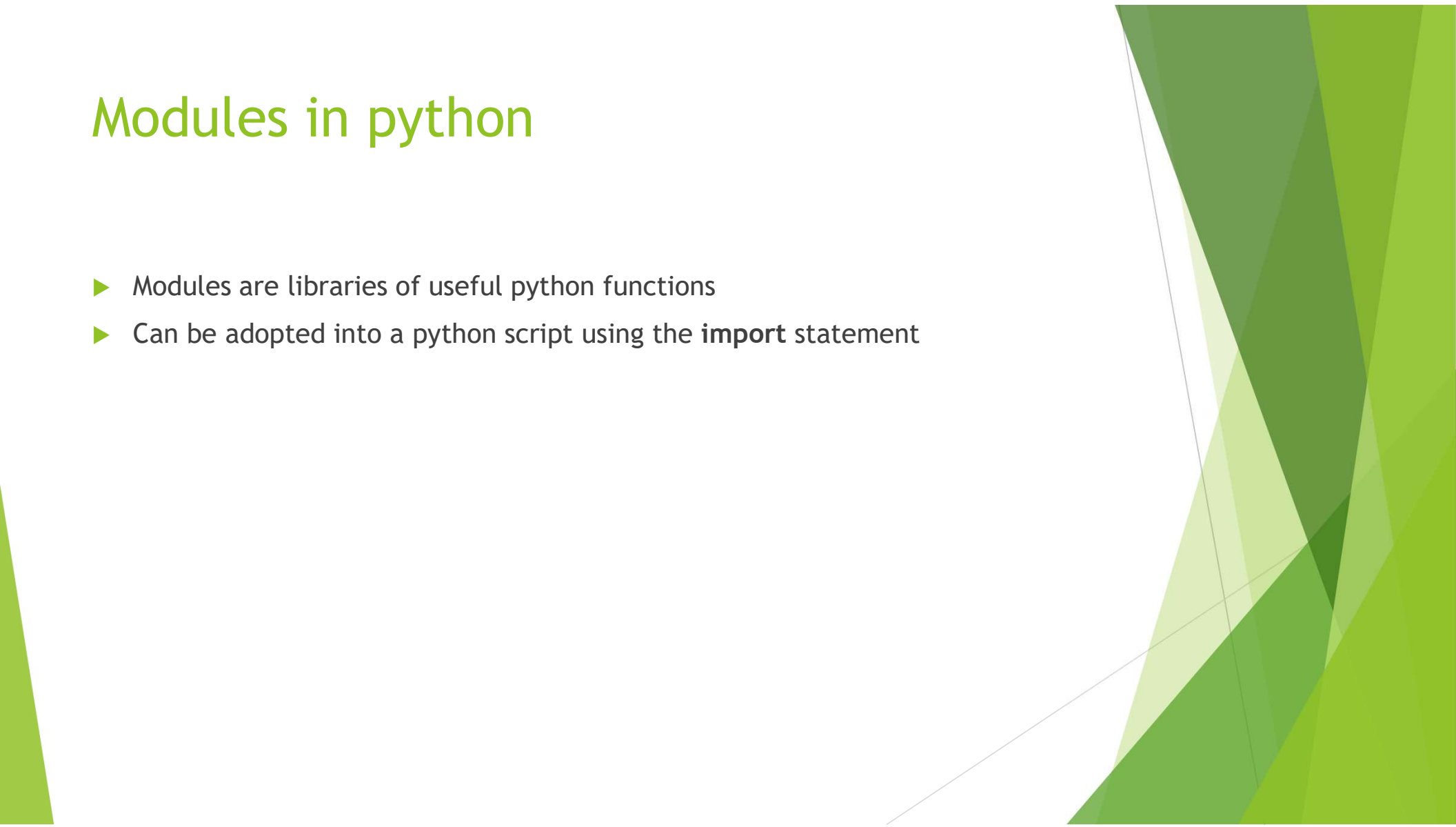
```
while <condition>:
    <statements>
```

User Defined Functions



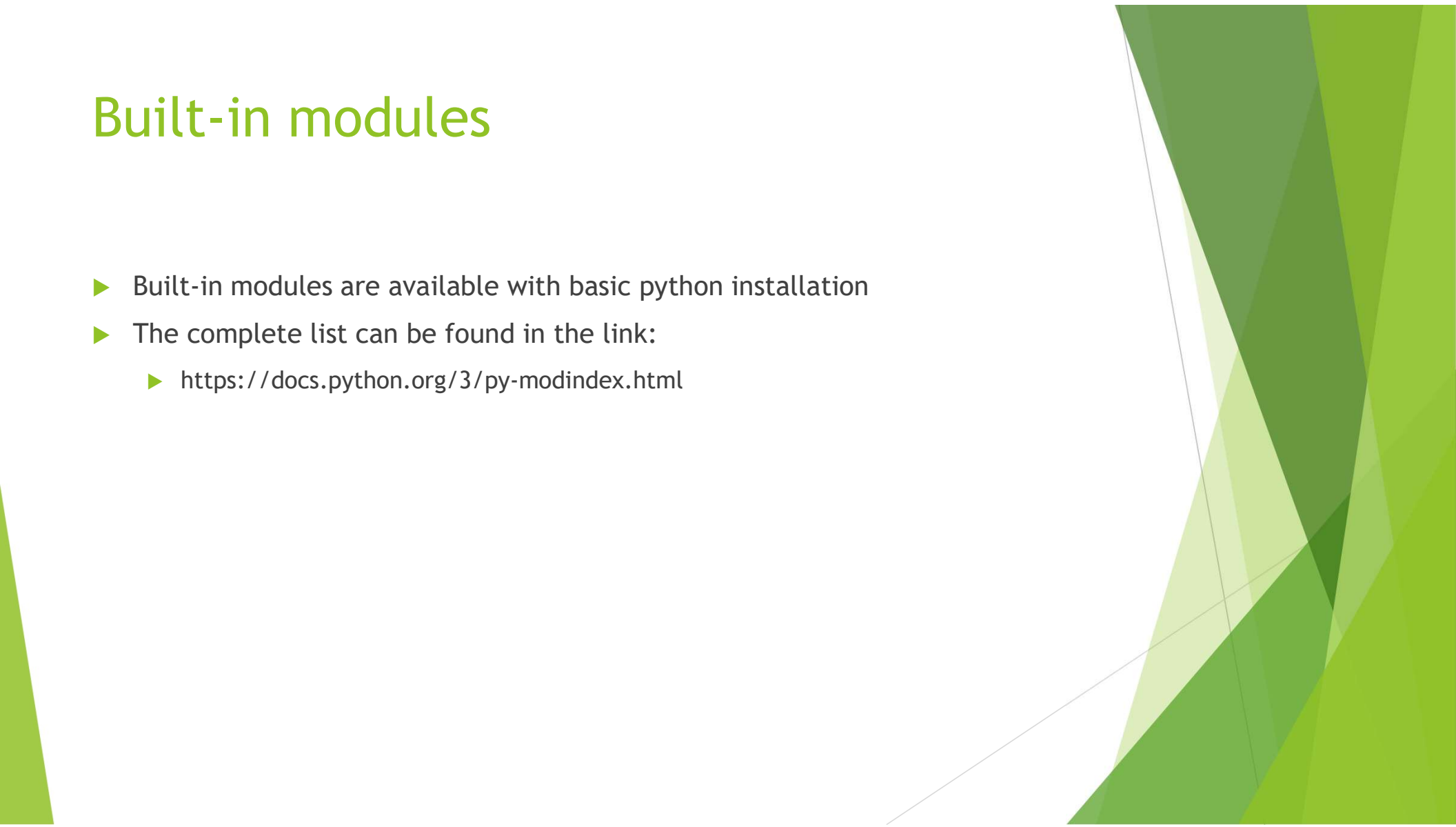
Modules in python

- Modules are libraries of useful python functions
- Can be adopted into a python script using the **import** statement



Built-in modules

- Built-in modules are available with basic python installation
- The complete list can be found in the link:
  - https://docs.python.org/3/py-modindex.html

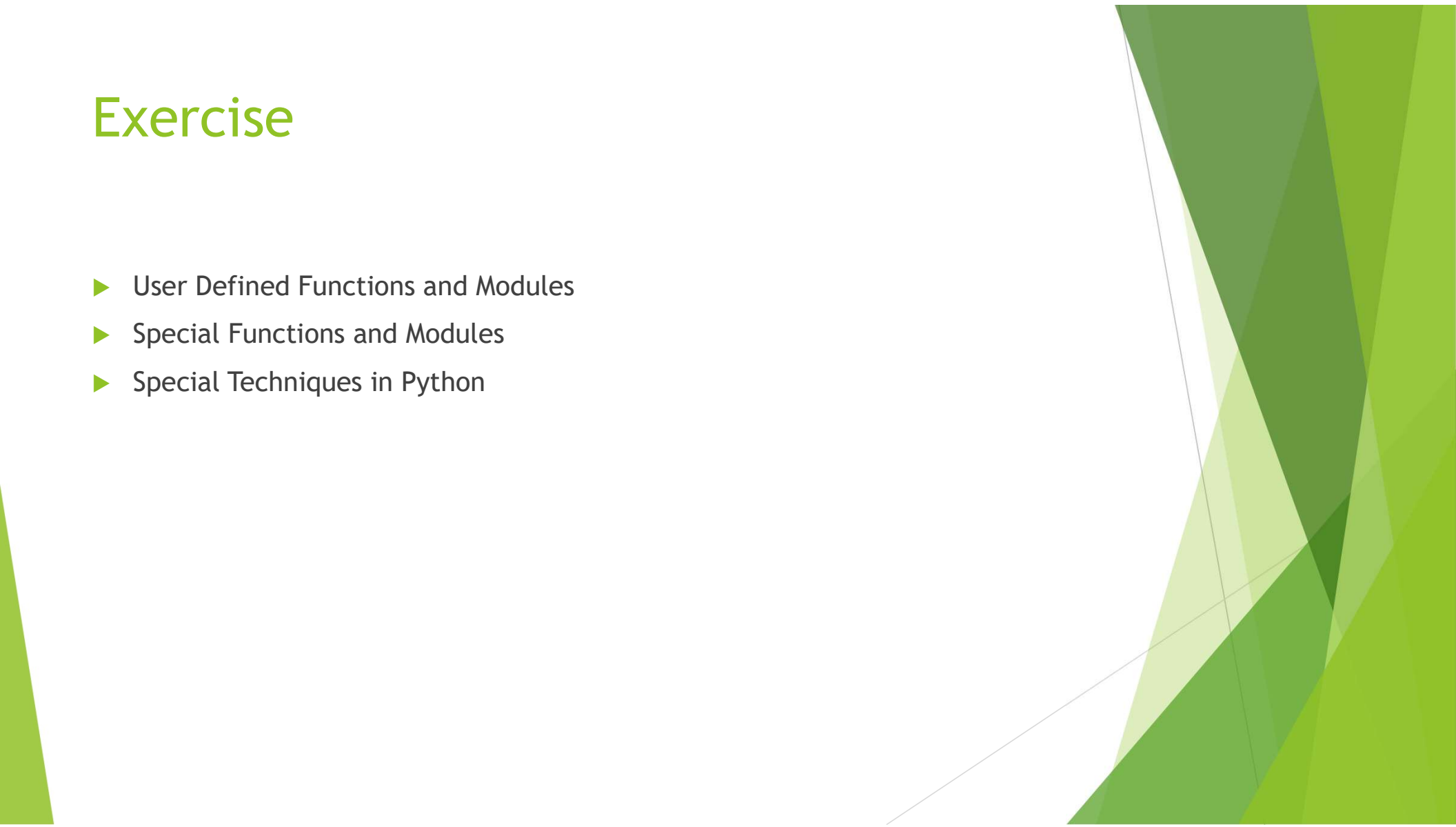


Third party modules and installing them

- Python modules published by other python users across the world
- Installing them:
  - Search for the module based on a given context
  - Install using the **pip install** command
  - Can be uninstalled using **pip uninstall** command
- Example:
  - Matplotlib
  - NumPy
  - Pandas

Exercise

- User Defined Functions and Modules
- Special Functions and Modules
- Special Techniques in Python

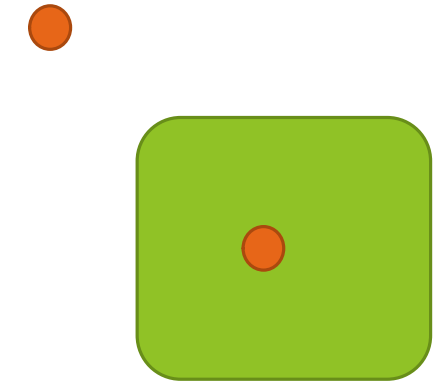


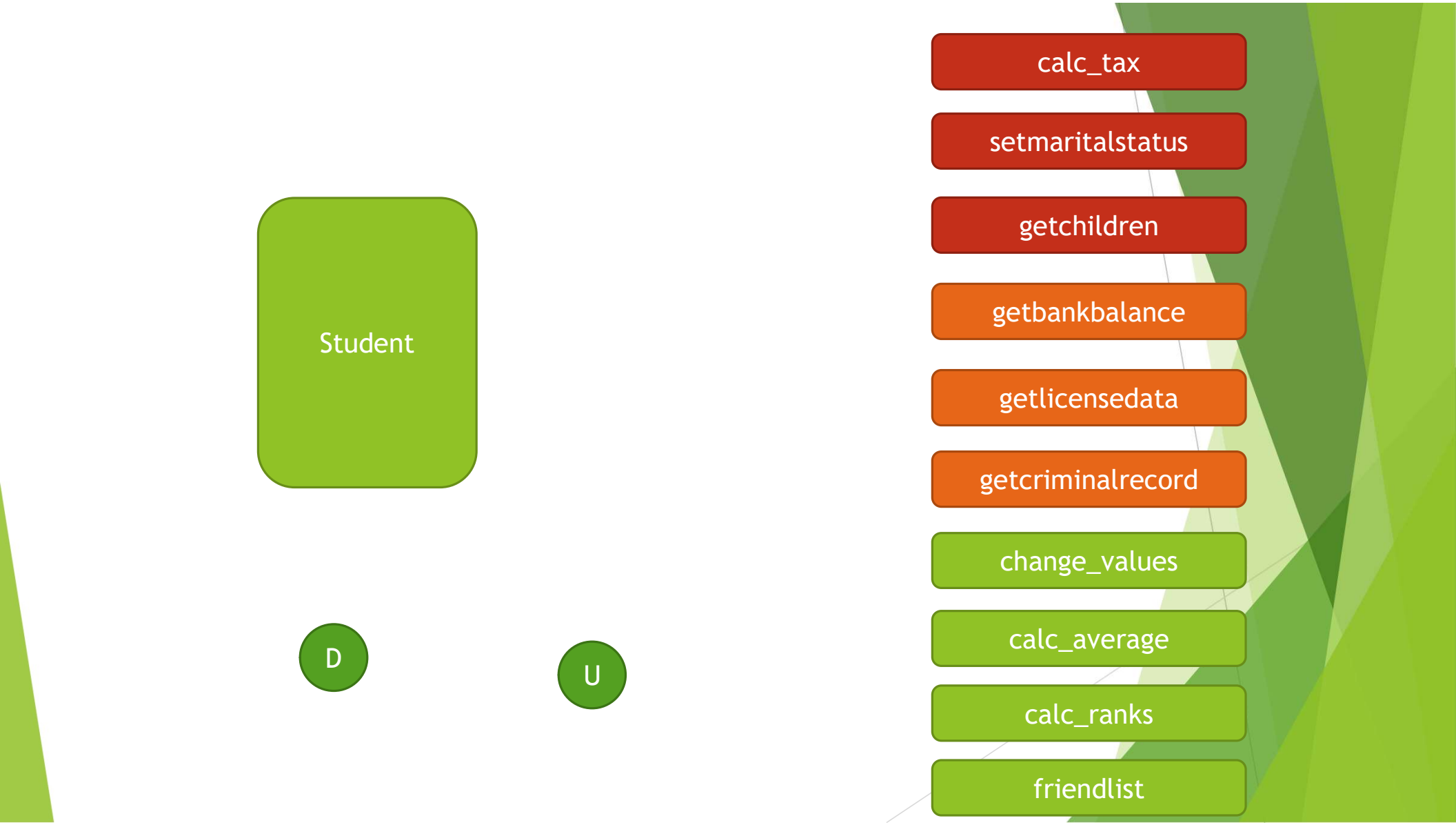
Practice

- Student Report Generation
- Design the Word Jumble Game
  - Discuss the disadvantages

OOP

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

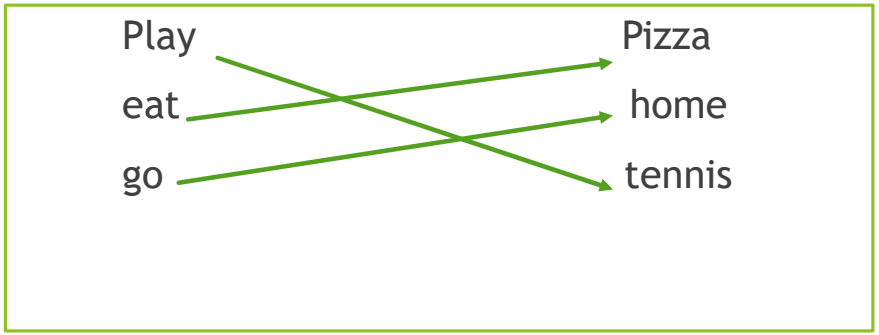


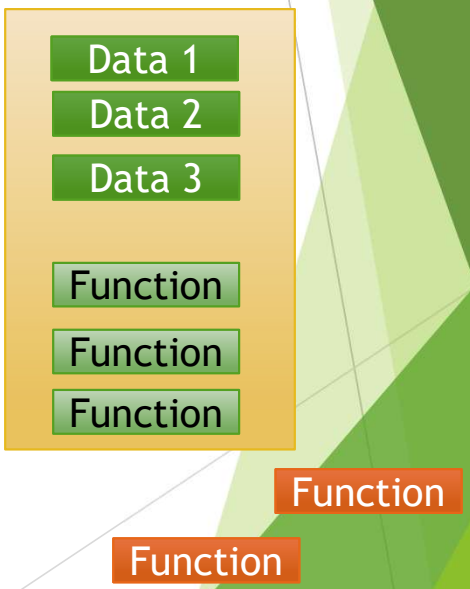

Student
D
U
calc_tax
setmaritalstatus
getchildren
getbankbalance
getlicensedata
getcriminalrecord
change_values
calc_average
calc_ranks
friendlist

But why “object-oriented” ness in programming

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

Ram

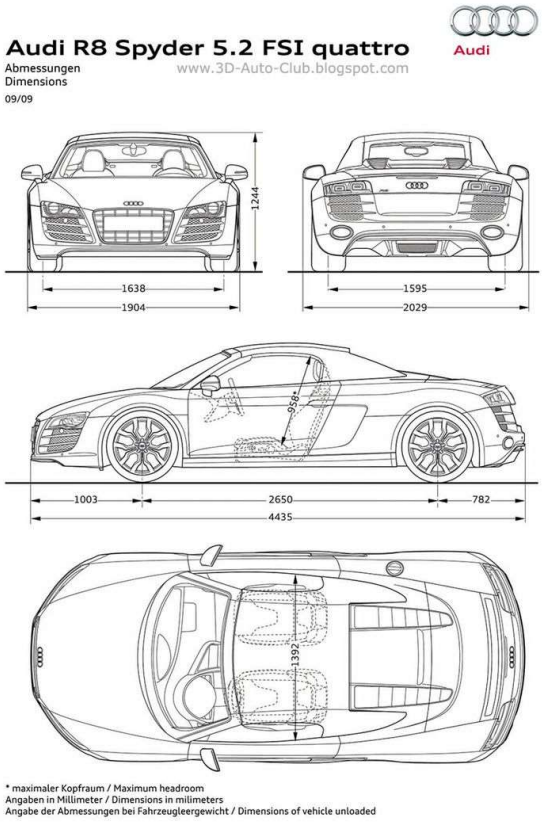




Function

Function

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

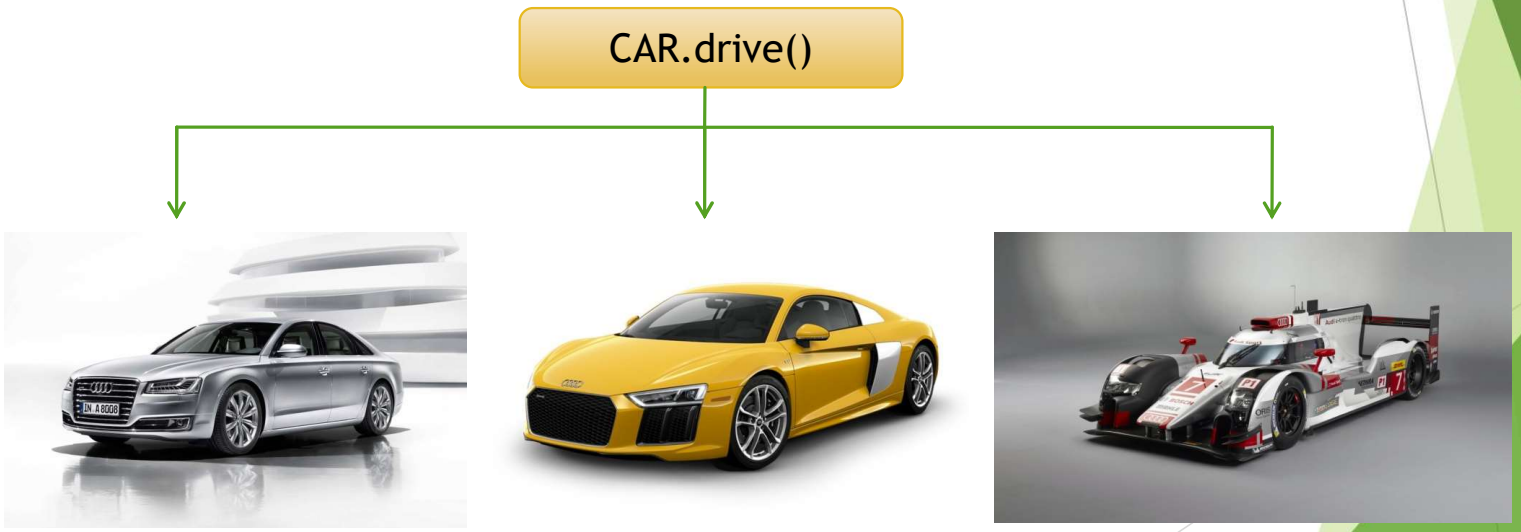
You can drive a car by operating the steering wheel and pedals, without knowing how the engine works. Similarly, you use an object through its methods. The implementation is hidden.

Inheritance

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

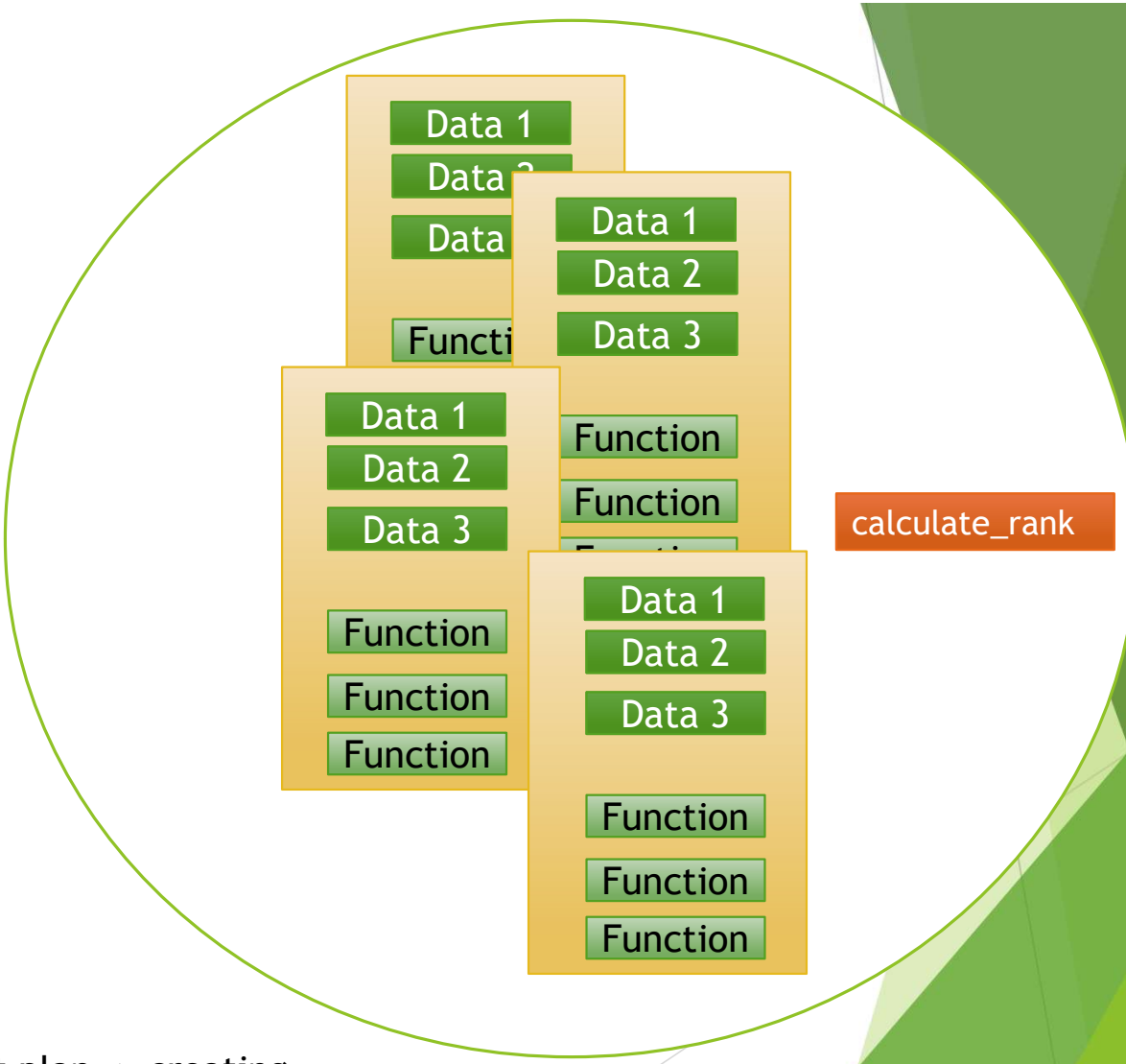
Polymorphism

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

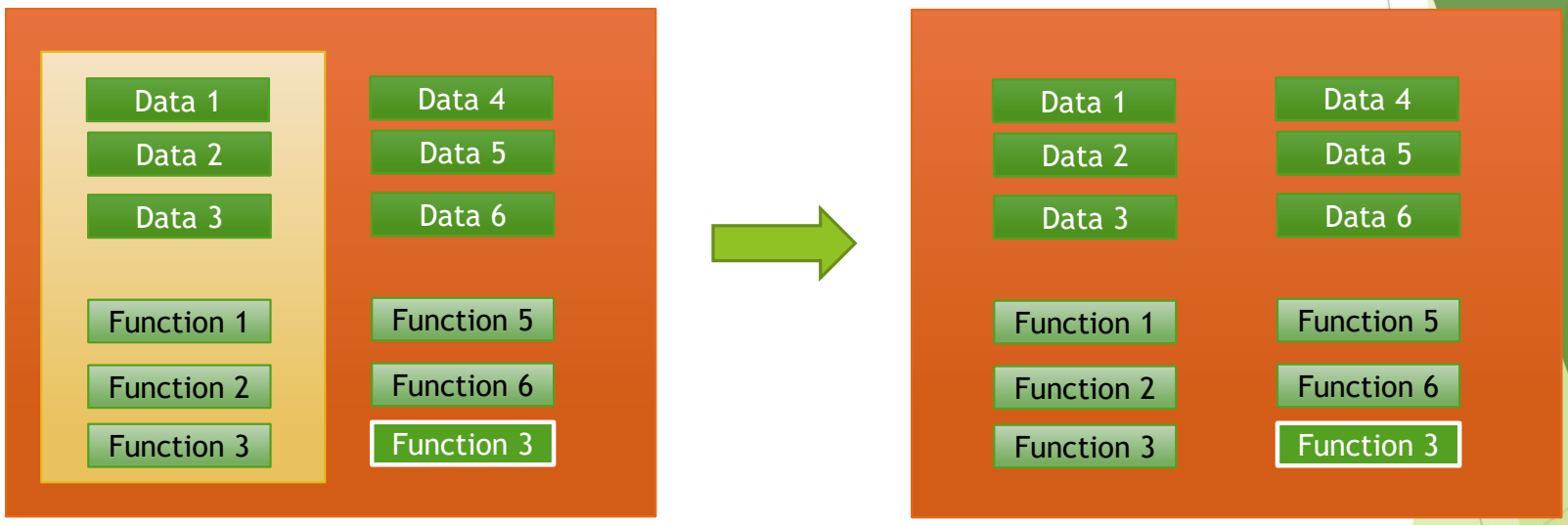


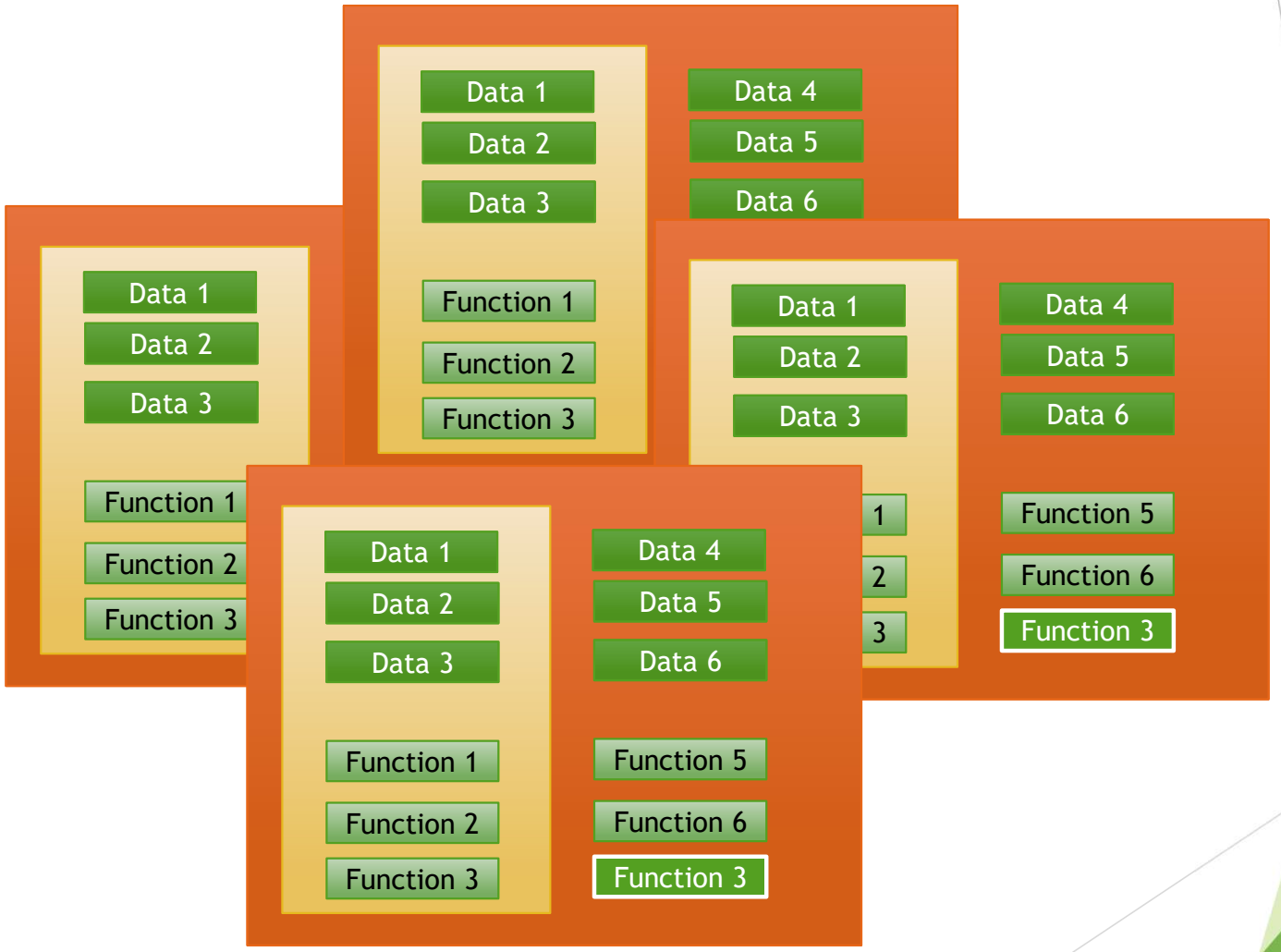
Let’s discuss about driving and maintenance tasks for cars…

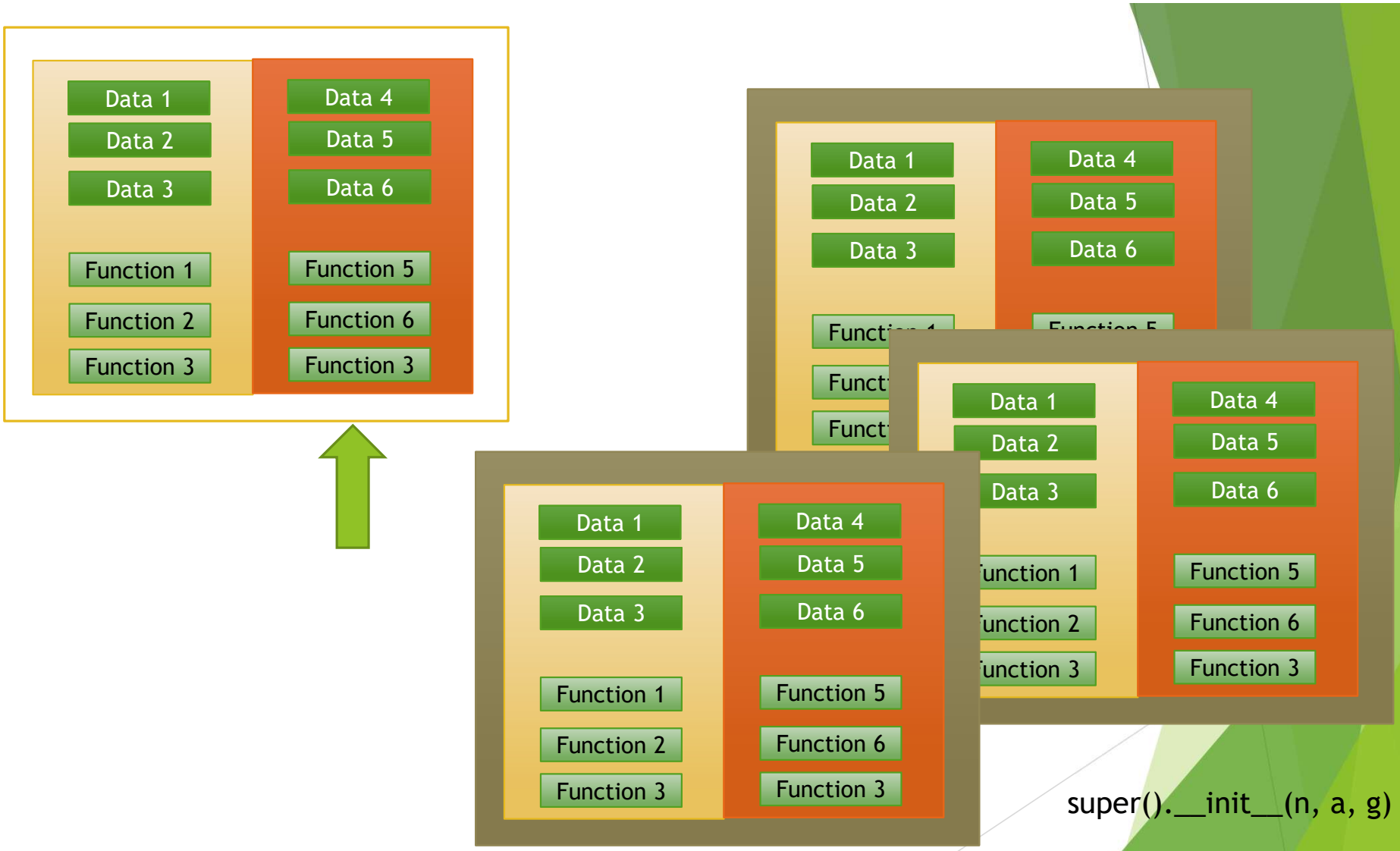
Plan -> class



Implement plan -> creating objects


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

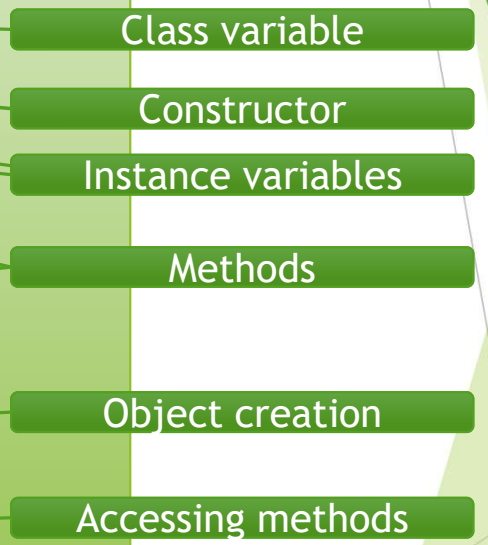

Data 1
Data 2
Data 3
Function 1
Function 2
Function 3
Data 4
Data 5
Data 6
Function 5
Function 6
Function 3



super().__init__(n, a, g)

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.

Testing the Applications

- Using pytest module

Practice

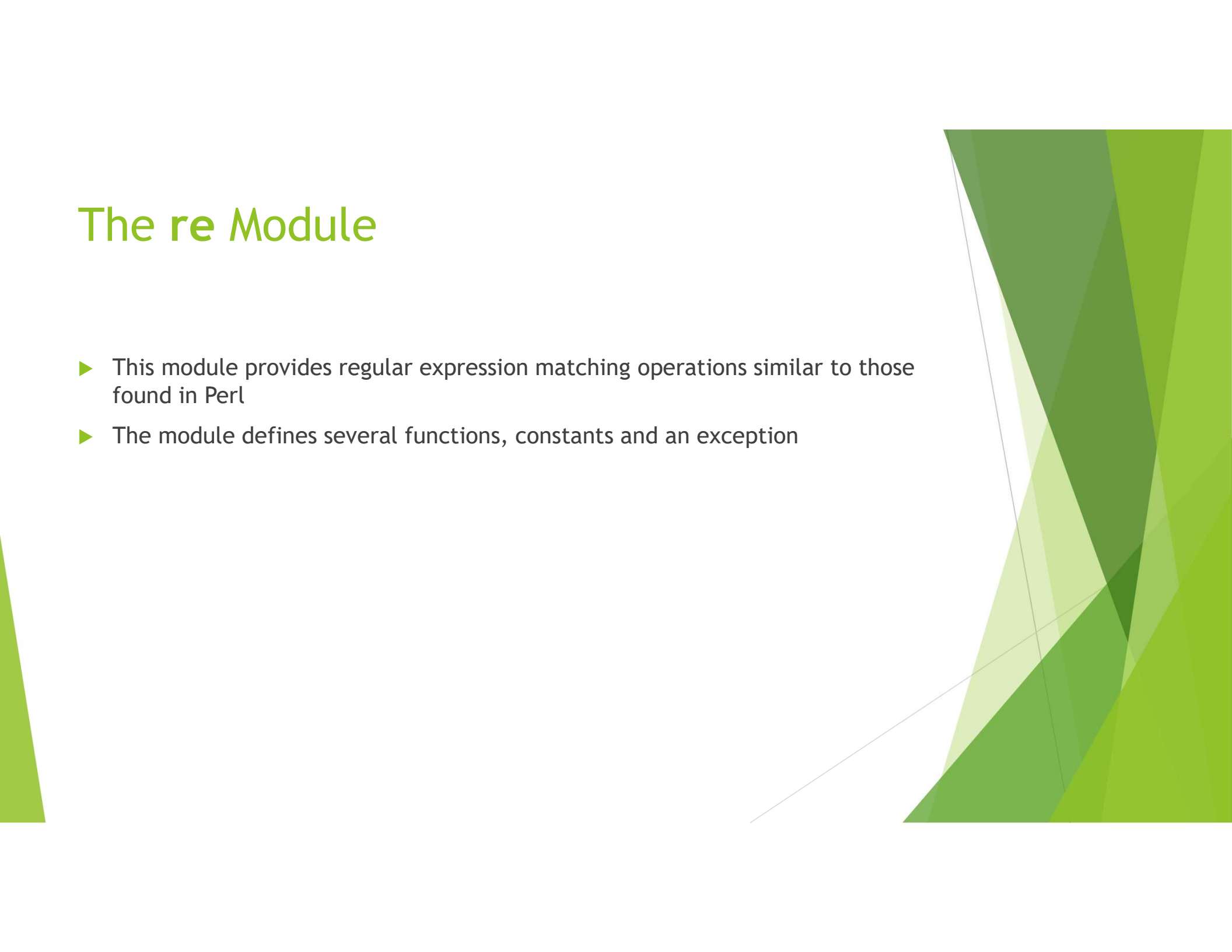
- Re-do the Word Jumble Game in OOP style
  - Are the disadvantages resolved?
- Design the TODO Application

Assignment

- Design a Cards Game Application

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.

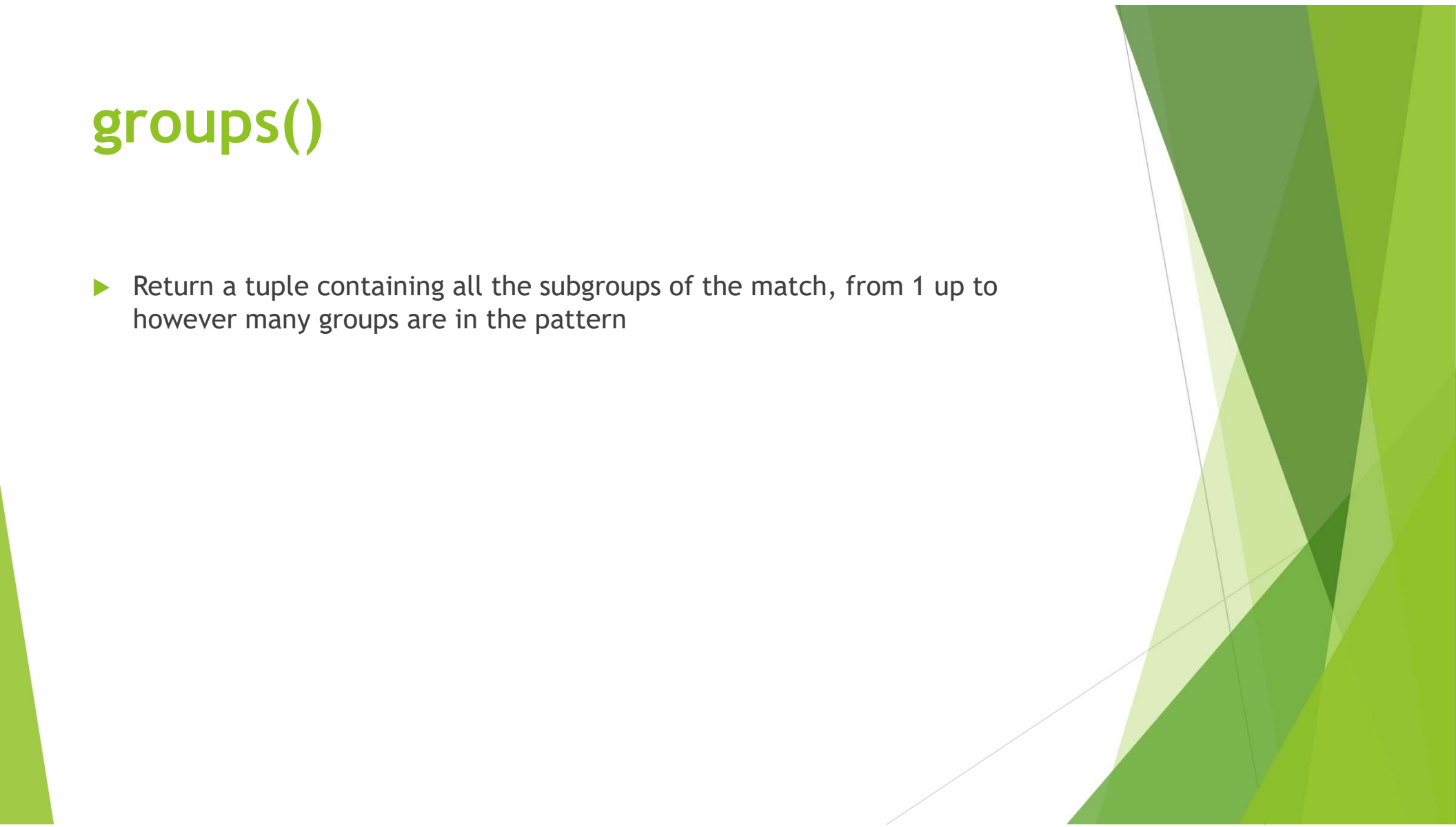
```
>>> email = "tony@tiremove_thisger.net"
>>> m = re.search("remove_this", email)
>>> email[:m.start()] + email[m.end():]
'tony@tiger.net'
```

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

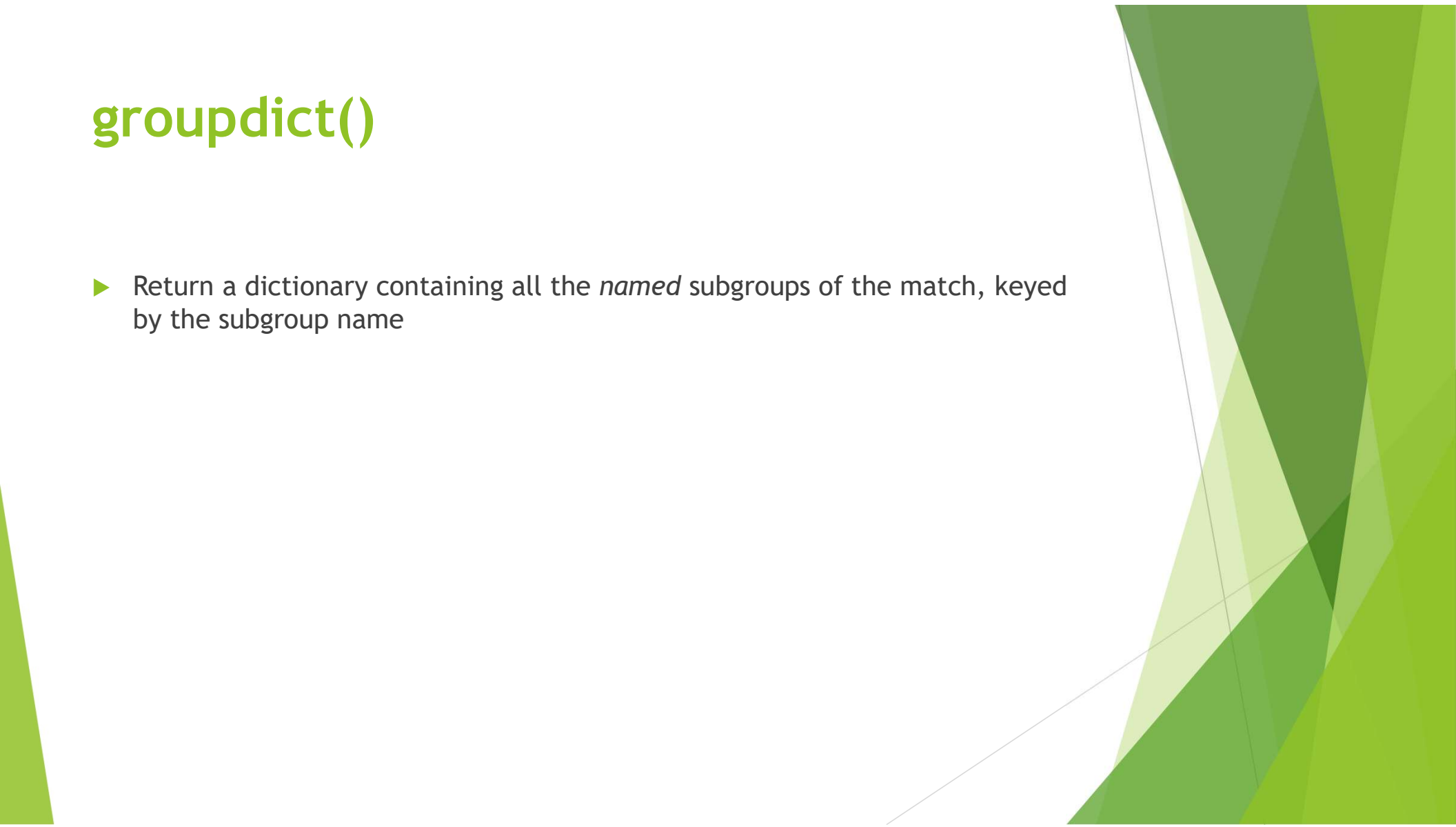
groups()

- 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'
```

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]

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.

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'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)
```

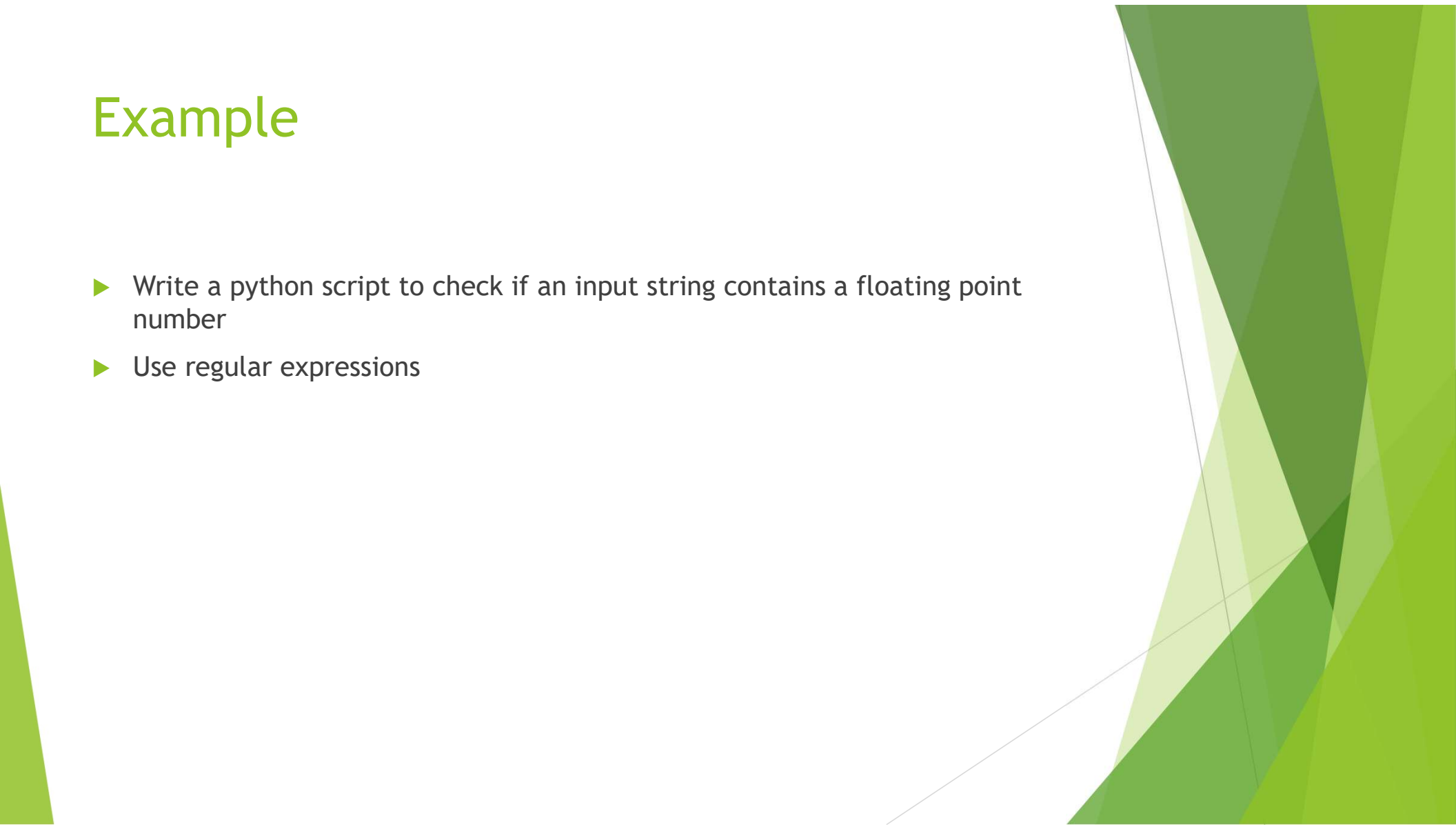
```
a5555b matched in 'a5555b'
a55555b matched in 'a55555b'
a555555b matched in 'a555555b'
```

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

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

Example

- 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'
```

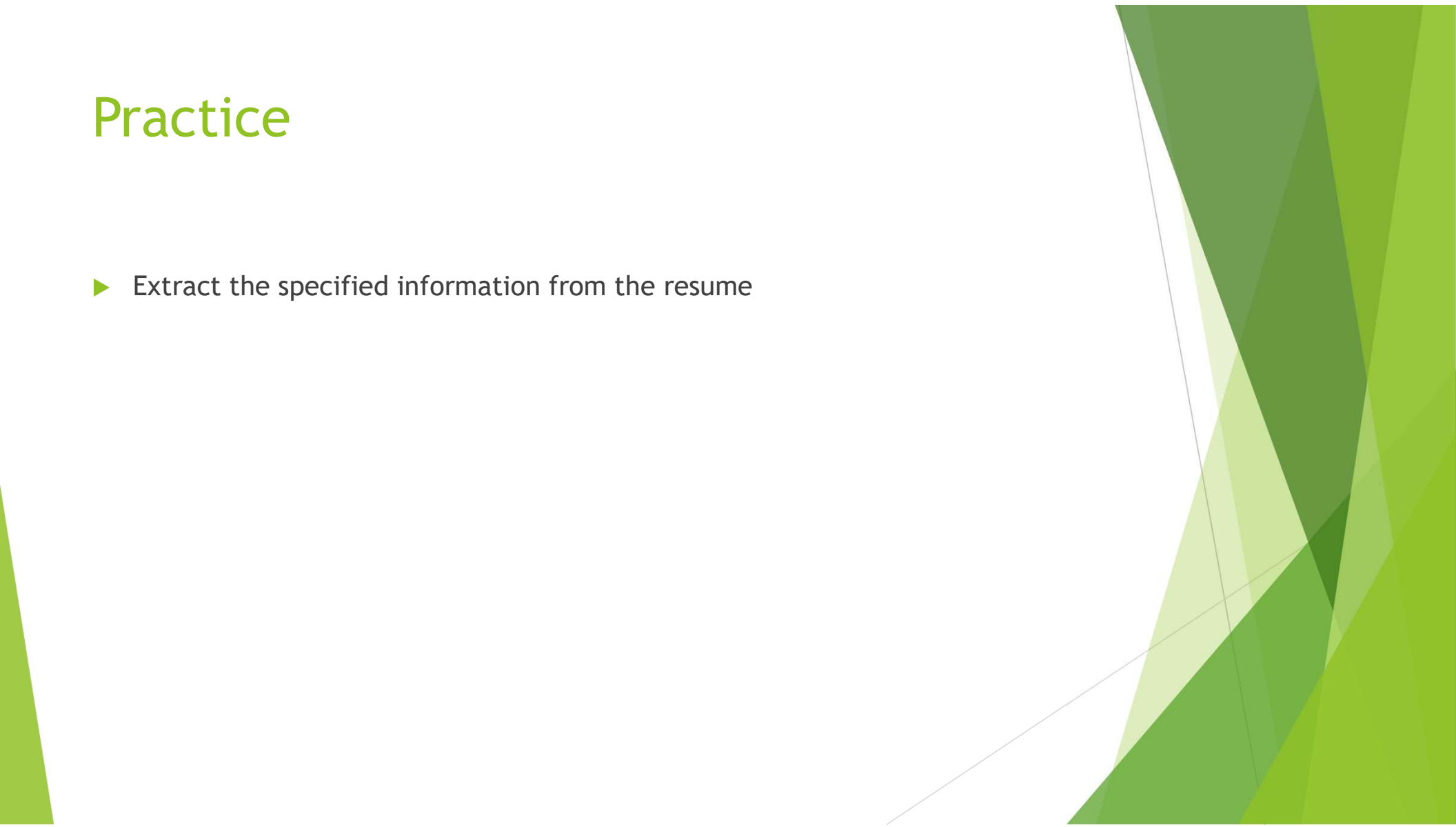
```
======= RESTART: E:/Python27/mindful_examples/regex/floating_point.py =======
37.0 degree centigrade is equal to +98.6 farhenheit
Trying to find a floating point numbers in the statement...
Following numbers were found ['37.0', '+98.6']
```

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

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

Practice

- Extract the specified information from the resume



Handling Exceptions

- Use **try.. except** block