KIT100 PROGRAMMING PREPARATION

Lecture Nine:

Creating and using Classes

Lecture Objectives

- Procedural Programming
- Object-Oriented Programming
  - Understanding Classes
  - Considering the Parts of a Class
  - Creating a Class
  - Using the Class in an Application
  - Exception handling

Procedural Programming

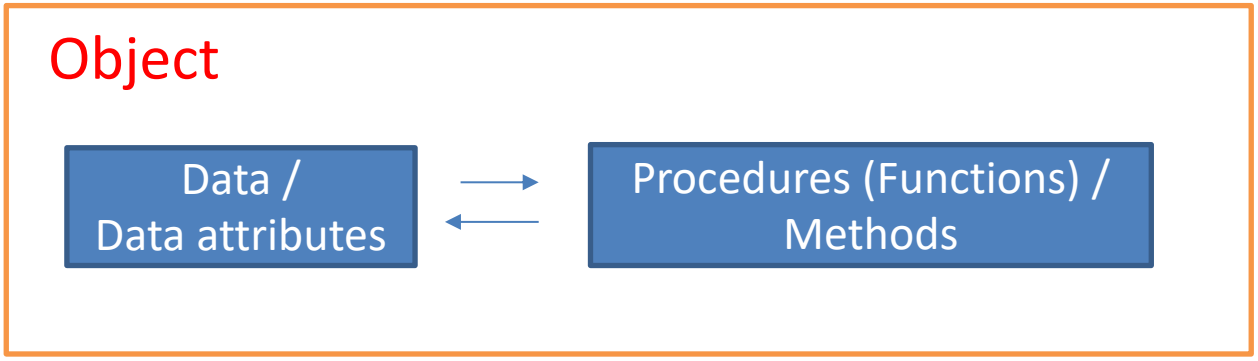
- Up to now, we have been using *procedural programming*

- Procedural programming divides our program into **reusable 'chunks'** called procedures, also known as **functions**.
  - Procedures typically operate on data items that are separated from the procedures.
    - function (data item 1, data item 2)
  - Data items are commonly passed from one procedure to another.
    - function1 (data item 1, data item 2); function2 (data item 1, data item 2)
  - Focus: defining functions, variables/data, parameter passing … to create procedures.

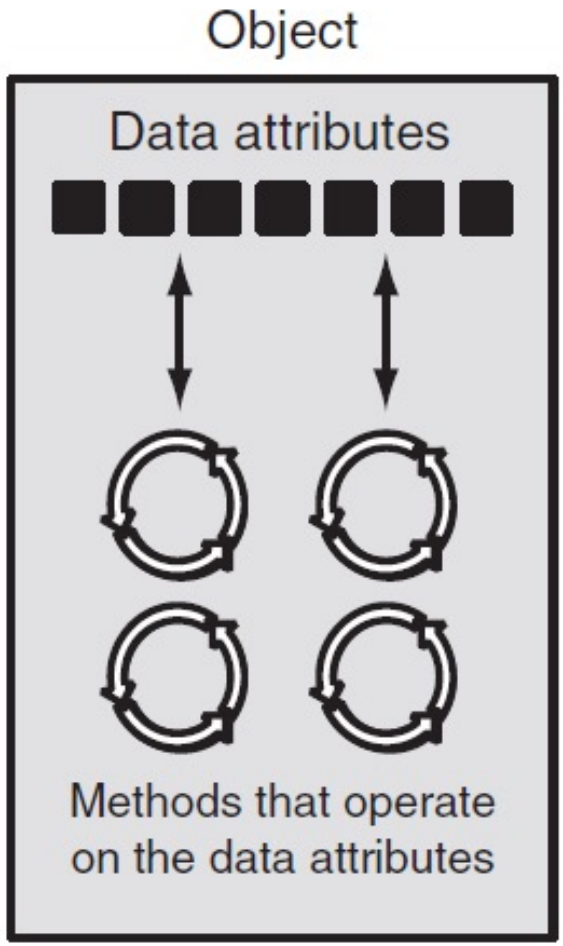
Object Orientated Programming

Terminology

- Object-Orientated programming is centered on *objects*.
- Object: an **entity** that contains **data and procedures**
  - Inside an object,
    - **data** is known as *data attributes*
    - **procedures** (functions) are known as *methods*
  - Methods perform operations on the data attributes



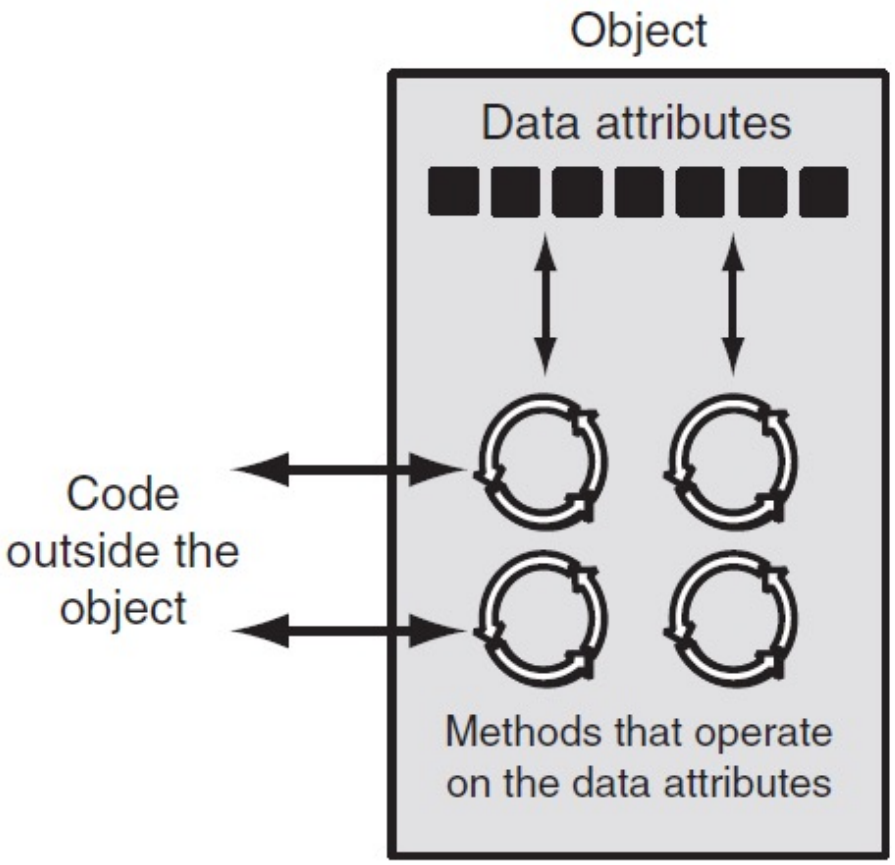
Object-Oriented Programming



Some Methods:

- "internal" use only - private methods.

Object-Oriented Programming



Some Methods:

- "internal" use only - private methods.
- use by code "outside" the object - public methods.
- "special outside" code (specifically subclasses of the parent class) – protected methods. [Data hiding - the next slide]

Terminology (cont.)

- Data hiding:
  - an object's data attributes are hidden from code outside the object.
- Access is restricted to the object's methods.
  - Protects from accidental corruption.
  - Outside code does **not** need to know the internal structure of the object, we only need to know how to interact with the object (via the methods).
- Object reusability:
  - an object can be used in different programs
    - we don't have to write or rewrite code that the object represents, we can just use them as is.

Procedural programming vs. Object-Oriented Programming

Task: Process and print the grade of students (name, score).

- ***Procedural programming***

Dictionary type (key: value)

```
# 1.Procedural programming

std1 = { 'name': 'Michael', 'score': 98 }
std2 = { 'name': 'Bob', 'score': 81 }

def print_score(std):
    print('%s: %d' % (std['name'], std['score']))

# Procedural print
print_score(std1)
print_score(std2)
```

Output:

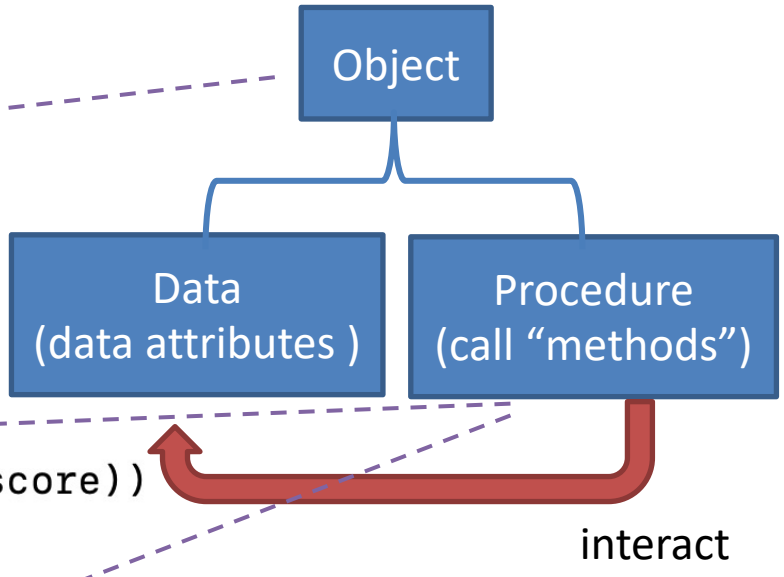
```
Michael: 98
Bob: 81
```

- ***Object-Oriented Programming (OOP)***
  - NOT consider how to execute the program (procedural programming)
  - Need to think about the data attributes of objects (name, score…)
  - How?
    - 1) create an object (including data attributes and methods);
    - 2) send the method (e.g.,'print_score' function) to defined objects.

Task: Process and print the grade of students (name, score).

- *Object-Oriented Programming (OOP)*

```
# 2.Object-Oriented Programming (OOP)

class Student(object):

    def __init__(self, name, score):
        self.name = name
        self.score = score

    def print_score(self):
        print('%s: %d' % (self.name, self.score))

# OOP print: Method

bart = Student('Bart Simpson', 59) # 'Student' is a object
lisa = Student('Lisa Simpson', 87)

bart.print_score()
lisa.print_score()
```



Output:

```
Bart Simpson: 59
Lisa Simpson: 87
```

Do not worry,
will explain the details of this code

An everyday example of an object

- **Data attributes**:
  - define the **state** of an object.
  - example: a clock object would have `second`, `minute`, and `hour` data attributes.
- **Public methods**:
  - methods that allow **external code** to manipulate the object's data attributes.
  - example: `set_time`, `set_alarm_time`
- **Private methods**:
  - used by the object itself for the object's inner workings i.e. not available or visible to external code

Understanding ClassES

- **Classes** make working with Python code more convenient.
  - Classes are used as *containers* for building data and functionality together.
  - Classes can then be reused as is by other code.
  - Compared with other programming languages, Python's class mechanism adds classes with a minimum of new syntax and semantics.

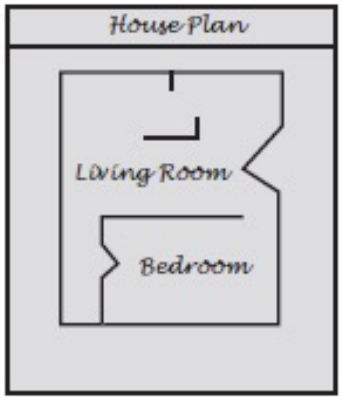
Understanding ClasseES

- **Class**:
  - Source code that specifies the data attributes and methods of an object.
    - it defines how objects are created.
    - like a blueprint of a house.

- **Instance**:
  - an object that is created from a class blueprint.
    - there can be **many** *instances* of **one** class.
    - like a specific house built according to the blueprint.
      - e.g. an `Animal` class defines animals, we can then create many instances of it, 2 dogs, a cat, 76 elephants etc.

Understanding ClassES

Blueprint that describes a house

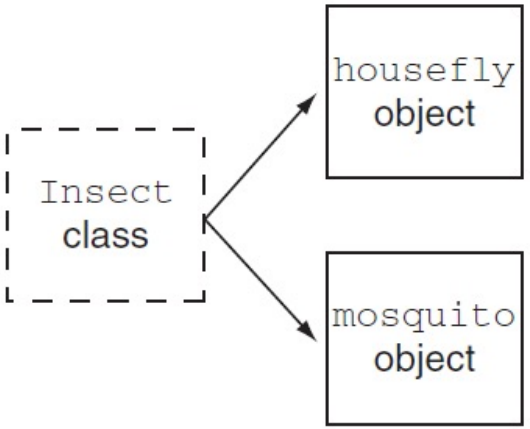


Instances of the house described by the blueprint

Understanding ClassES

Example:

The `Insect` class describes the data attributes and methods that a particular type of object may have.



The `housefly` object is an instance of the `Insect` class. It has the data attributes and methods described by the `Insect` class.

The `mosquito` object is an instance of the `Insect` class. It has the data attributes and methods described by the `Insect` class.

Conclusion:

The `housefly` and `mosquito` objects are instances of the `Insect` class

Key Python terms for ClassES

Key Python terms for class (Object-Oriented Programming)

- **Class**:
  - Defines the structure and data ("attributes") that will be used for objects.

- **Object (or *Instance*)**:
  - A collection of data and methods (that manipulate the data) **created** from the class definition.

- **Class variable**:
  - Provides a (public/shared) storage location used by **all** instances of a class.

- **Instance variable:**
  - Provides a (private) storage location used by each instance of a class.

Key Python terms for ClassES

Key Python terms for class (Object-Oriented Programming)

- **Data Member**:
  - Refers to either a class variable or an instance variable.

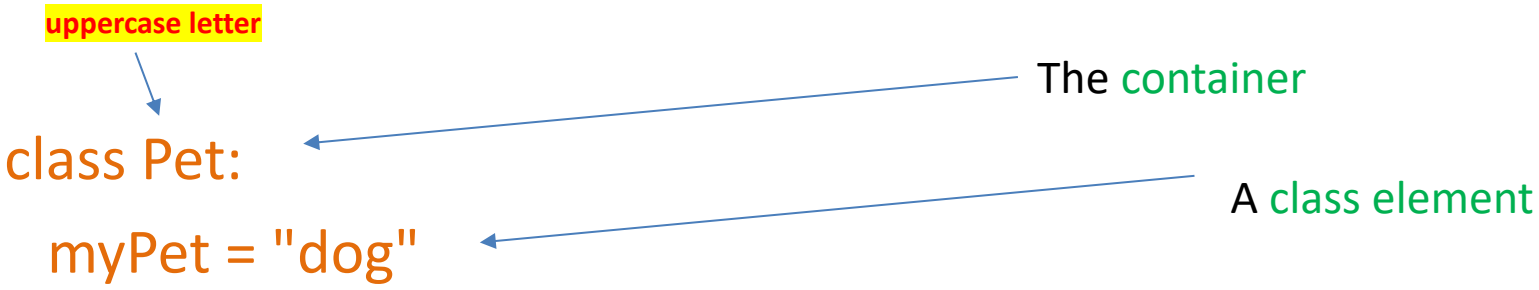
- **Method**:
  - Another name for a function (defined in the class definition) that can manipulate or examine the class' data members.

- **Inheritance**:
  - A class can be defined by inheriting attributes and methods from a parent class (known as "subclassing") before it then makes specialised modifications.

Considering the parts of a Class

- A class follows a specific format.
  - The class begins with a container.
  - Then come the class elements.

- A class doesn't need to be complex, in fact we can just create the container, and one class element.



The class

- Data structures:
  - like lists and strings, are extremely useful, but sometimes they aren't enough (or become too complex to keep track of implementation).

- Task: consider how could we keep track of **pets** initially **without** using the class:
  - A list might work, we could use the first element of the list as the pet's name, and the second element of the list as the pet's species. *(the next page - an example attached)*
  - But how do we know which element is supposed to be which?
    - We need to remember the meaning of which element is
      - i.e. the order is important – what if we forget, or don't document it? Someone using our pet code later might make a mistake…

Lists with arbitrary content

```
"""
Title: Pet List
Author: Zehong Jimmy Cao
Date: March 2020
Purpose: To provide an example of a list with Pet details
"""

petList = ['Polly','Parrot','Fido','Dog'] # a list of pet details.

print(petList)
print(petList[0]) # Even-numbered indices are pet names
print(petList[1]) # Odd-numbered indices are pet species
```

This is "easy" to get wrong!

Output

```
['Polly', 'Parrot', 'Fido', 'Dog']
Polly
Parrot
>>>
```

Class definitions

- **Class definition**: set of statements that define a class's methods and data attributes.
  - Format: begin with `class ClassName:`
    - Class names should start with an **uppercase letter**
  - Method definitions are like any other python function definition, except one parameter is **always** required:
    - `self` parameter – references the instance (a specific object) that called the method. When you define a method, the first parameter must be '`self`'.
    - If we have several objects, we need to know which one is calling (using) the method. `self` allows us to do this (python sets the variable automatically).
    - Note `self` is not a reserved word, but it's used by convention for this purpose.

Class definitions

- **Initializer method**: automatically executed (once) when an instance of the class is created
  - Initializes object's data attributes and automatically assigns the `self` parameter to the object that was just created.
  - Format: `def __init__ (self):`
    - Starts with **two** underscore characters, followed by the word `init`, followed by **two** more underscore characters
  - Python relies on this method or *constructor* to perform tasks.
    - such as assigning values to any instance variables that the object will need when it springs into existence.
  - It's usually the first step implemented (written) **in** a class definition.

Go back to the example @ page 9.

Creating the class definition

```
"""
Title: Creating a class definition pt 2
Author: Zehong Jimmy Cao
Date: March 2021
Purpose: To provide an example of class definition in Python
"""


class Pet:
```

Good programming style:
Always start a class name with a **capital letter**!

Reason: normal variables start with lower case
– we can instantly know whether we're dealing with a class or a variable by the case of the first letter!

Creating the class definition

```
"""
Title: Creating a class definition pt 2
Author: Zehong Jimmy Cao
Date: March 2021
Purpose: To provide an example of class definition in Python
"""

class Pet:

    def __init__(self, name, species):
        self.name = name
        self.species = species
```

Always put 'self'

Putting `self.` *before a variable name, such as self.species.*

- *It means we're referring to an 'instance' variable – it makes the variable accessible to any code in the class – it means it has global (class) scope.*

`__init__` is a special method name.

- It's the pre-defined method name for a class' ***constructor***.
- When you are creating an object (instantiation), this method is run **once**.
- It tells Python if there's anything special it needs to do (e.g. set values for instance variables) as the object is created. Here when we create a Pet instance (object) we're specifying the Pet's name and species.

Creating the class definition

```
class Pet:

    def __init__(self, name, species):
        self.name = name
        self.species = species


    def getName(self):
        return self.name

    def getSpecies(self):
        return self.species
```

- The class's data attributes (some or all) are *exposed* to external code via the methods.
- Here we're able to see what the `name` and `species` values are using the `getName` and `getSpecies` methods.
- Outside code **never** refers to the `self.name` or `self.species` variables directly (they're part of the 'internal workings' of the class)

Creating the class definition


```
"""
Title: Creating a class definition pt 2
Author: Zehong Jimmy Cao
Date: March 2021
Purpose: To provide an example of class definition in Python
"""

class Pet:

    def __init__(self, name, species):
        self.name = name
        self.species = species


    def getName(self):
        return self.name

    def getSpecies(self):
        return self.species

    def __str__(self):
        return ("%s is a %s") % (self.name, self.species)
```

Another special method name.

*Usage: if you ask an object to **print** itself, this method "__str__" is run.*

Using Classes

How to **use** the classes (@ Python IDLE window) ?

Python 3.8.1 Shell

```
Python 3.8.1 (v3.8.1:1b293b6006, Dec 18 2019, 14:08:53)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: /Users/czh513/Desktop/KIT001/.....week9/Petsp26.py
>>> from Petsp26 import Pet
>>> JimmyPet = Pet("Polly", "Parrot")
>>> print(JimmyPet.getName())
Polly
>>> print(JimmyPet.getSpecies())
Parrot
>>> print(JimmyPet)
Polly is a Parrot
>>>
>>> samsPet = Pet("Kylo", "Cat")
>>> print(samsPet.getName())
Kylo
>>> print(samsPet.getSpecies())
Cat
>>> print(samsPet)
Kylo is a Cat
>>>
```

File name

Class name

Import the path

Summary of Using Classes

- **To create a new instance of a class** call the *initializer method* and assign the result (which is a reference to an object) to a variable:
  - **Format**: *myInstance = ClassName()*
    - example
```
>>> samsPet = Pet ("Kylo", "Cat")
```
    - *Python implicitly calls the* *__init__* *method for you here*
      i.e. it does this: *myInstance = ClassName().__init()__*

equal

```
example >>> samsPet = Pet.__init__("Kylo", "Cat")
```

  - If your object *__init__* method requires data to set data attributes when the object is created, pass them as parameters
    Format: *myInstance = ClassName(data1,data2,etc)*
- **To "call" any of the class methods** using the newly created instance, use **dot notation** (.)
  - **Format**: *myInstance.methodName(data1,data2,etc)*

```
>>> samsPet = Pet ("Kylo", "Cat")
>>> samsPet.getName()
'Kylo'
>>> samsPet.getSpecies()
'Cat'
```

Using Classses

Execution statement:

```
myPet = Pet("Polly", "Parrot")
```

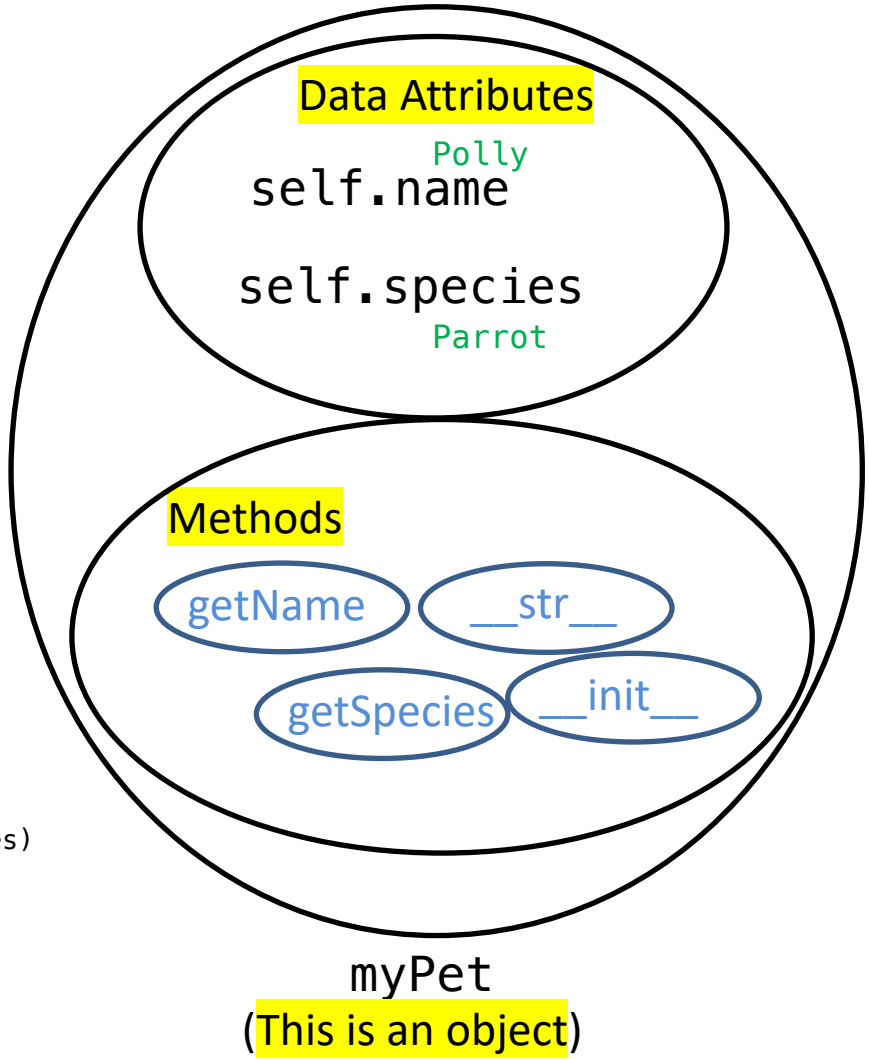
This runs the `__init__` method

name is Polly

specie is Parrot

auto-set

```
class Pet:

    def __init__(self, name, species):
        self.name = name
        self.species = species

    def getName(self):
        return self.name

    def getSpecies(self):
        return self.species

    def __str__(self):
        return ("%s is a %s") % (self.name, self.species)
```



Using classes

More on `self`:

- The **defined** `__init__` method (as well as the other methods in the `Pet` class) have this `self` variable parameter e.g.

  ```
  def __init__(self, name, species):
  ```

  but that when we **use** the method e.g.

  ```
  polly = Pet(self,"Polly", "Parrot"),
  ```

  you only have to pass in two values (*name* and *species*) instead of three.
- Why?
  - We **don't have to provide** the `self` parameter ourselves.
  - This is a special behaviour of Python: **when you call a method (use it)** of an instance, Python automatically figures out what `self` should be (from the instance) and implicitly passes it to the function. In the case of `__init__`, Python first creates `self` and then passes it in.

The __str__ method

- It is very common for programmers to equip their classes with a method that returns a string containing some representation of the object's state (the values of the object's data attributes at a given moment).
  - In Python, you give this method the special name `__str__` .

- *How to display the object's state?*
  - Use **`__str__`** method.
  - You usually **do NOT directly** call the `__str__` method yourself
    - Instead, it is automatically called when you try to convert the object to a string – this typically happens when you:
      - try to print the object (i.e. when you pass it as a parameter to the `print()` function); or
      - when the object is passed as an argument to the `str()` function

Creating more Classes

```
"""
Title: Creating a class definition
Author: Jimmy
Date: March 2021
Purpose: To provide an example of class definition in Python
"""

class ExampleClass:
    greeting = ""

    def __init__(self, name = "there"):
        self.greeting = name + "!"

    def sayHello(self):
        print("Hello %s"%(self.greeting))
```

A default value!

Output

How to use it?

```
= RESTART: /Users/czh513/Desktop/KIT001/
p32.py
>>> from exampleClass_p32 import ExampleClass
>>> myInstance = ExampleClass()
>>> myInstance.sayHello()
Hello there!
>>> myInstance = ExampleClass("Jimmy")
>>> myInstance.sayHello()
Hello Jimmy!
>>>
```

.....

Variable scope – normal vs instance variables

Variable scopes – normal variables vs. "instance" variables

Example 1:

```
class ExampleClass:
    def doAdd(self, value1=0, value2=0):
        mySum = value1 + value2

    def printMySum(self):
        print("The value of mySum is",mySum) # Output – ERROR!
```

*mySum is **only defined** in the doAdd method.*

mySum is **not defined** here!

Example 2:

```
class ExampleClass:
    def doAdd(self, value1=0, value2=0):
        self.mySum = value1 + value2

    def printMySum(self):
        print("The value of mySum is",self.mySum)
# Output can execute!
```

self.mySum** makes the variable an instance variable – its scope is now the **entire object

*- i.e. it's defined and visible **everywhere** inside the object.*

Variable scope – normal vs instance variables

Example 1:

inst_p35.py - /Users/czh513/Desktop/KIT001_2021/1 Lecture/week9/w9 lecture examples/inst_p35.py (3.8.1)

```
"""
Title: Creating a instance variables
Author: Jimmy
Date: March 2021
Purpose: To provide an example of instance variables in Python
"""


class ExampleClass:
    def doAdd(self, value1=0, value2=0):
        mySum = value1 + value2
        print ("The sum of %d plus %d is %d" %(value1, value2, mySum))

    def printMySum(self):
        print("The value of self.mySum is", mySum)
```

Output

```
>>> from inst_p35 import ExampleClass
>>> myThing = ExampleClass()
>>> myThing.doAdd()
The sum of 0 plus 0 is 0
>>> myThing.printMySum()
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    myThing.printMySum()
  File "/Users/czh513/Desktop/KIT001_2021/1 Lecture/week9/w9 lecture examples/inst_p35
.py", line 16, in printMySum
    print("The value of self.mySum is", mySum)
NameError: name 'mySum' is not defined
>>>
```

Variable scope – normal vs instance variables

inst_p35.py - /Users/czh513/Desktop/KIT001_2021/1 Lecture/week9/w9 lecture examples/inst_p35.py (3.8.1)

```
""" Example 2:
Title: Creating a instance variables
Author: Jimmy
Date: March 2021
Purpose: To provide an example of instance variables in Python
"""

class ExampleClass:
    def doAdd(self, value1=0, value2=0):
        self.mySum = value1 + value2
        print ("The sum of %d plus %d is %d" %(value1, value2, self.mySum))

    def printMySum(self):
        print("The value of self.mySum is", self.mySum)
```

Output

Ln: 14

```
>>>
>>> from inst_p35 import ExampleClass
>>> myThing = ExampleClass()
>>> myThing.doAdd()
The sum of 0 plus 0 is 0
>>> myThing.printMySum()
The value of self.mySum is 0
>>> myThing.doAdd(2,3)
The sum of 2 plus 3 is 5
>>> myThing.printMySum()
The value of self.mySum is 5
>>>
```

myThing refers to an instance (an object) of the `ExampleClass` class.

call the `doAdd` method on the `myThing` object.

call the `printMySum` method on the `myThing` object.

Dealing with Errors

- Most application code of any complexity has **errors** in it!
  - Errors can occur and sometimes they do NOT provide you with any sort of notification of that error.
  - Errors aren't always consistent.
- Errors in **running programs** are called ***exceptions.***
  - **Detecting *exceptions*** is called ***exception handling***.
  - Don't confuse these types of errors with syntactic errors – exceptions are errors that occur *as the program is running* (they are also called *runtime* errors)

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

```
>>> while True:
...     try:
...         x = int(input("Please enter a number: "))
...         break
...     except ValueError:
...         print("Oops!  That was no valid number.  Try again...")
...
```

Knowing why Python doesn't understand you

- We often get frustrated with programming languages and computers because sometimes it seems they are being dense deliberately!

- Remember:
  - Even though we use a *Programming Language* to communicate with the computer, the computer will never 'know what you mean'.
  - It will always treat your code literally, without understanding you might have meant something different to what you coded.

Ways of dealing with errors

Steps:

1. Catching exceptions
   - detect where errors might occur, and add code to "filter them out", like catching fish in a fishing net.
   - it prevents your program from crashing (and stopping!)
2. Handling exceptions
   - add code to take special actions (usually remediation tasks) when errors are caught (detected)
3. Raising (or *throwing*) exceptions
   - add code that **deliberately** causes or forces specific types of errors to occur!
   - there will hopefully then be exception catching and handling code that then kicks in to act accordingly.

```
>>> while True:
...     try:
...         x = int(input("Please enter a number: "))
...         break
...     except ValueError:
...         print("Oops!  That was no valid number.  Try again...")
...
```

Why do errors occur?

- Errors occur in many cases when the developer makes assumptions that simply aren’t true.
- Python doesn’t understand the concepts or good or bad data.
- Python isn’t proactive or creative.

Errors of time – compile time errors

- A compile time error occurs when you ask Python to run the application.
  - Compile-time errors are the easiest to spot and fix.
  - The appearance of a compile-time error should tell you that other typos or omissions could exist in the code.

- Compile time error
  - Includes syntax errors,
  - but also errors that occur due to e.g. missing class files that are referred to in your code (which occur when we are trying to link modules (classes) written elsewhere.

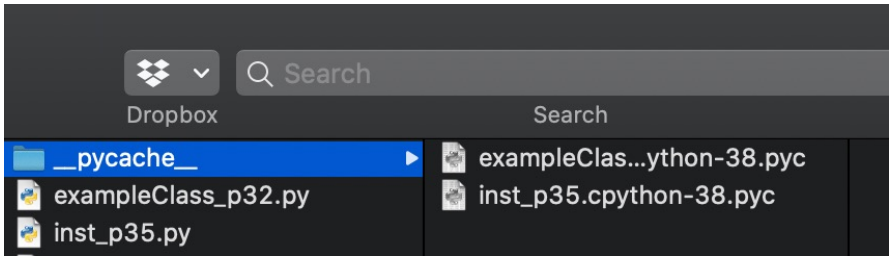
Errors of time – runtime errors

- A runtime error occurs as Python **executes your source code**
  - Runtime errors come in several different types.

- Most runtime errors produce exceptions that can be caught, but some cannot be caught
  - or even if they are caught, we can't necessarily fix the issue in the exception handling code.
  - These can cause program instability and misbehavior and invariably the program crashes.

Reminder of error types

- Understanding the error types help you locate errors faster. We discussed these early in the semester
- Three types:
  - Syntactical
  - Semantic
  - Runtime (exceptions errors *as the program is running*)



When you run a program in python, the interpreter compiles it to bytecode first and stores it in the __pycache__ folder.

Syntactical errors

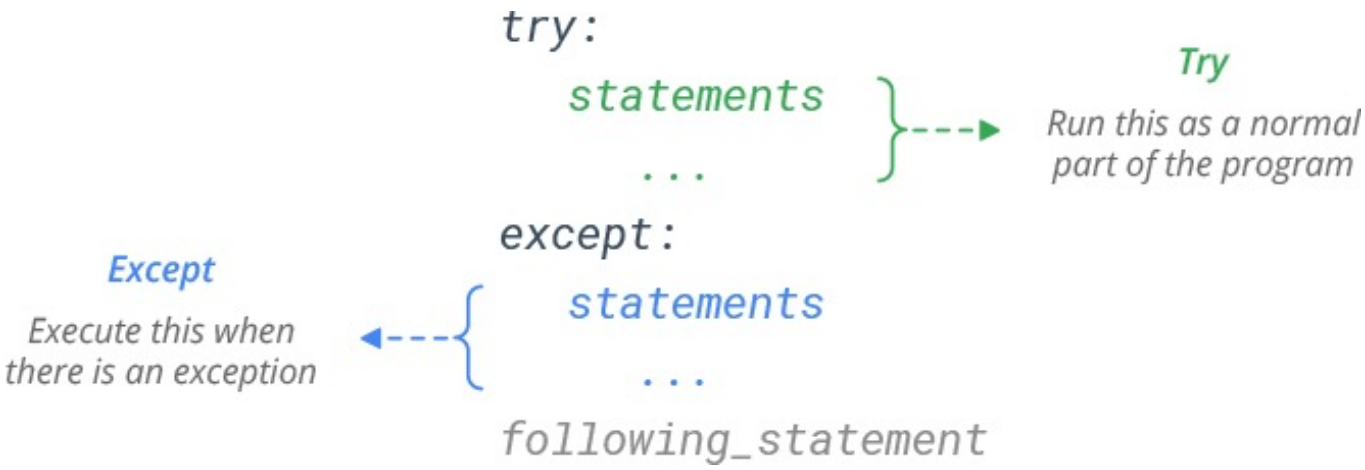
- Whenever we make a typo of some sort, we create a syntactical error.
- Most syntactical errors occur at compile time
  - in Python's case, the interpreter scans the source code prior to execution, and the interpreter points them out for you (IDLE in our case).
- Syntactical errors prevent the application from running.
- Easiest to fix!

Semantic errors

- Off-by-one errors in loops are an example of semantic errors.
  - E.g., List [start, end-1]
- Semantic errors occur because the intended meaning behind a series of steps used to perform a task is wrong.
- There are different tools we can use with Python to perform debugging when looking for semantic errors.
- Hardest to fix!

Runtime (Exception) handling

- We can use exception handling to provide a robust solution to when users enter something unplanned.
- To handle exceptions we must tell Python what we want to do and provide code to perform the handling task.
- **Exception handling** in Python is done using exceptions that are caught in `try` blocks and handled in `except` blocks.



Runtime (Exception) handling

Example 1

runtime_p47.py - /Users/czh513/Desktop/KIT001/

```
try:
    x = 1/0
except:
    print('Something went wrong.')
```

```
= RESTART: /Users/czh513/Desktop/KIT001
y
Something went wrong.
>>>
```

Example 2

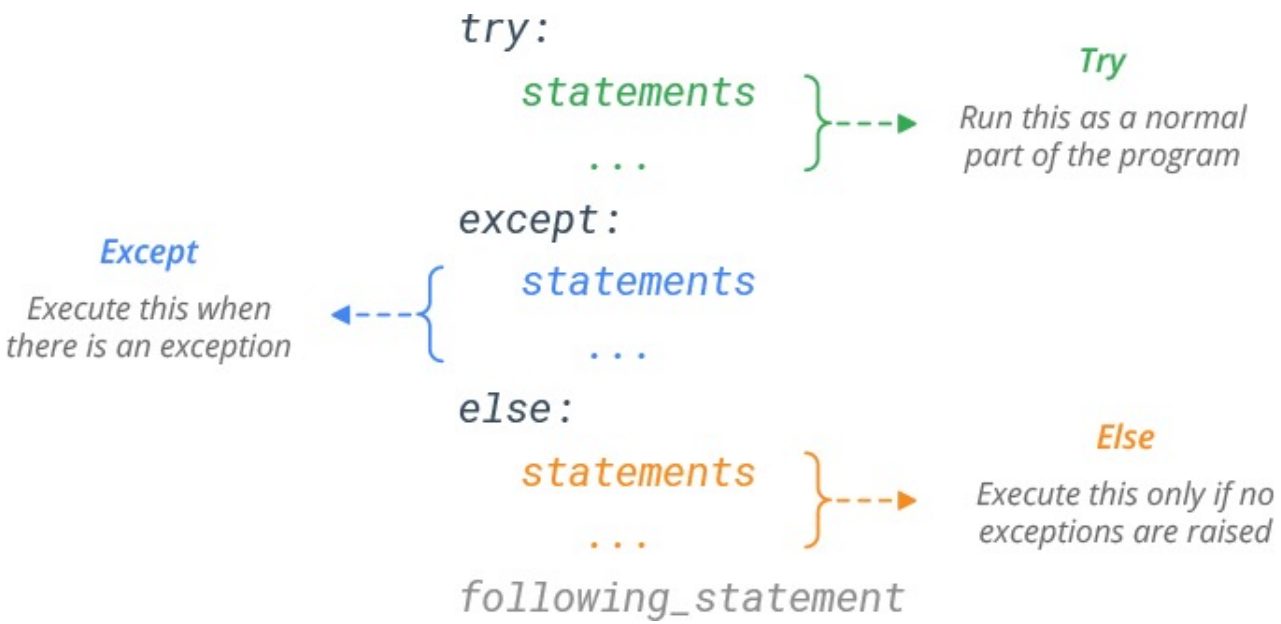
```
try:
    x = 1/0
except ZeroDivisionError:
    print('Attempt to divide by zero')
except:
    print('Something else went wrong')
```

```
= RESTART: /Users/czh513/Desktop/KIT001/Teach
y
Something went wrong.
Attempt to divide by zero
>>>
```

if … else with exception handling

The Else Clause:

- The try…except block has an optional else clause.
- The **else clause is executed** only if no exceptions are raised.



if … else with exception handling

```
# example 1
try:
    x = 1/1

except:
    print('Something went wrong')
else:
    print('Nothing went wrong')


# example 2
try:
    value = int(input("Enter a number:"))
    print("Well Done")

except ValueError:
    print("A dummy user!")
else:
    print('A good user!')
```

Output

```
= RESTART: /Users/czh513/Des
des/test_p49.py
Nothing went wrong
Enter a number:5
Well Done
A good user!
>>>
= RESTART: /Users/czh513/Des
des/test_p49.py
Nothing went wrong
Enter a number:Jimmy
A dummy user!
>>>
```

if … else with exception handling

- **An example (the pseudo code):**
  - Obtain an int input value from the user between 1 and 10.
  - Evaluate the user input to determine if the user entered an int value or not.
  - If the user did not enter an int, produce an exception message to the user.
  - If the user did enter an int and the value is between 1 and 10, display the number to the user.
  - If the int value is not between 1 and 10 display the message “The value you typed is incorrect!”.

```
# Demonstration of exception handling
valid = False


while (not(valid)):

    try:
        value = int(input("Type a number between 1 and 10: "))

    except ValueError:
        print("You must type a number between 1 and 10!")

    else:
        valid = True
        if (value > 0) and (value <= 10):
            print("You typed: %s" % value)
        else:
            print("The value you typed is incorrect!")
```

Output

```
= RESTART: /Users/czh513/Desktop/KIT001/
des/errorDemo_p49.py
Type a number between 1 and 10: 5
You typed: 5
>>>
= RESTART: /Users/czh513/Desktop/KIT001/
des/errorDemo_p49.py
Type a number between 1 and 10: 12
The value you typed is incorrect!
>>>
= RESTART: /Users/czh513/Desktop/KIT001/
des/errorDemo_p49.py
Type a number between 1 and 10: Hello
You must type a number between 1 and 10!
Type a number between 1 and 10: 7
You typed: 7
>>>
```

Some of the common exception error types

- `IOError`: If a file cannot be opened.
- `ImportError`: If python cannot find the requested module.
  - *e.g. you're trying to include a class and Python can't find the class source file.*
- `ValueError`: when a built-in operation or function receives an argument that has the *right type* but an *inappropriate value.*
  - *e.g.* `int()` *expects a string parameter, but the string itself might not contain numeric characters*
- `KeyboardInterrupt`: when the user hits the interrupt key.
  - normally `Control-C` or `Delete`
- `EOFError`: when one of the built-in functions hits an end-of-file condition (EOF), while trying to read more data, but there is no more data allowing to read.
  - e.g. `input()`.

```
try:
    check = input ("would you like to continue?: ")

except EOFError:
    print ("Error: EOF or empty input!")
```

Some of the common exception error types

- KeyboardInterrupt

```python
try:
    while True:
        print("Infinite loop!")
except KeyboardInterrupt:
    print("You interrupted the loop!")
else:
    print("Good user!")
```

Output

```
Python 3.8.1
Infinite loop!
Infinite loop!
Infinite loop!
Infinite loop!
Infinite loop!
Infinite loop!
Infinite loop!
Infinite loop!
Infinite loop!
Infinite loop!
Infinite loop!
Infinite loop!
Infinite loop!
Infinite loop!
Infinite loop!
Infinite loop!
Infinite loop!
Infinite loop!
Infinite loop!
Infinite loop!
Infinite loop!
Infinite loop!
Infinite loop!
Infinite loop!
Infinite loop!
Infinite loop!
Infinite loop!
Infinite loop!
Infinite loop!
Infinite loop!
Infinite loop!
Infinite loop!
Infinite loop!
Infinite loop!
Infinite loop!
Infinite loop!
Infinite loop!
Infinite loop!
Infinite loop!You interrupted the loop!
>>>
```

Control + C to stop

Questions?