Programming Paradigms

What are they?

- Guidelines for designing, organizing, and executing code.
- Distinct methods of writing, with unique advantages and disadvantages.
- They are **not** mutually exclusive, and languages can support multiple paradigms.

What does this mean in practice?

Choose the best parts of each paradigm for your project. The Object Oriented style keeps programs organized as they grow in size. The Functional style is hard to learn, but makes code easier to understand.

Functional Programming

Declarative -> Functional

- *Easy to test and understand, due to being relatively free of side effects, such as:*
  - Global variables, which are visible to anything in the environment. If they change, they can cause unpredictable behavior in functions which rely on them.
- *The outcomes of programs are based on composing pure functions.*
  - Pure functions return the same output given the same input. They are self contained and don't rely on external information.
  - There is no risk of mutable data or shared variables causing uncertainty. The function's behavior is predictable and stable.
  - When you combine two functions, like this: f(g(x)), that's function composition.

Functional Programming Sample

Functional

```
def checkForM(string):
        return "m" in string

presenceOfM = list(map(checkForM, ["Jameson", "Gondro",
"Gondromsomblom"]))


print presenceOfM

>>> => [True, False, True]
```

Non Functional

```
def checkForM(array):
        outputArray = []
        for string in array:
                If "m" in string:
                        outputArray.append(string)
        return outputArray

presenceOfM = checkForM(["Jameson", "Gondro",
"Gondromsomblom"])

print presenceOfM

>>> => [True, False, True]
```

*in Python

Object Oriented Programming

Imperative -> OOP

- *Bundles variables and functions into objects. The object acts as a toolbox of methods, for, modeling a real-world object, solving a specific problem, or holding information.*
    - This makes OOP code versatile, reusable, and easier to assemble and disassemble.
- *Multiple objects communicate with each other.*
    - Control flow depends on objects passing messages or exchanging information and methods.
    - Organize code into a hierarchical system of super-classes and sub-classes.

Object Oriented Sample

Object Oriented

```
class Person:
   def __init__(self, name):
      self.name = name

   def say_hi(self):
      print('Hello, my name is', self.name)

p = Person('Gumakull') # Gumakull is a distinct person that now exists and can be called any time
p.say_hi()

>>> "Hello, my name is Gumakull"
```

Non Object Oriented

```
Name = "Bonzoid"

def personSaysHi(Name):
        return "Hello, my name is %s" % Name

personSaysHi(Name)

>>> "Hello, my name is Bonzoid"

#Bonzoid is a facsimile. A hologram on the wall. Impermanent.
```

*in Python