**Key Terms**

* **Function**: A reusable block of code that performs a specific task. Defined using the def keyword.
* **Arguments**: Values passed into a function when calling it. Allows customizing behavior.
* **Variable Arguments**: Allows passing an arbitrary number of arguments to a function.
* **Keyword Arguments**: Arguments passed by name rather than position. Can have default values.

**Function Example**

def double(x):

    """Doubles a number"""

    return x \* 2

print(double(5)) # Prints 10

**Function with Arguments**

def full\_name(first, last):

    return first + " " + last

print(full\_name("John", "Doe")) # Prints John Doe

**Variable Arguments**

def sum\_all(\*numbers):

    sum = 0

    for n in numbers:

        sum += n

    return sum

print(sum\_all(1, 2, 3)) # Prints 6

**Keyword Arguments**

def greet(name, greeting="Hello"):

    print(greeting + ", " + name)

greet("John") # Prints Hello, John

greet("Mary", greeting="Hi") # Prints Hi, Mary

**Generator:** A type of iterable like lists or tuples but does not store the full sequence in memory at once. Uses yield to generate one item at a time

#counter generator

def counter(start=0):

    n = start

    while True:

       yield n

       n += 1

for i in counter(5):

    if i > 10:

        break

    print(i)

5

6

7

8

9

10

#Infinite Fibonacci sequence generator

def fib():

    a, b = 0, 1

    while True:

       yield a

       a, b = b, a + b

for n in fib():

   if n > 10:

     break

   print(n)

0

1

1

2

3

5

8

**Generator expression:** More compact syntax like list comprehensions for inline lazy generation, uses () instead of []

#Generator expression to get squares

nums = (x\*\*2 for x in range(10))

print(list(nums))

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

**Infinite sequence:** Generators can be infinitely recursive/iterative to model data streams

# Infinite random attack sequence

import random

attacks = ["kimura", "armbar", "triangle"]

def lazy\_random\_attacks():

    """Lazily yield random attacks forever"""

    while True:

        attack = random.choice(attacks)

        print("Yielding attack")

        yield attack

generator = lazy\_random\_attacks()

for \_ in range(5):

    print(next(generator))

Yielding attack

kimura

Yielding attack

armbar

Yielding attack

armbar

Yielding attack

triangle

Yielding attack

kimura

The key points:

* While loop to iterate forever
* Uses yield to lazily produce one attack at a time
* Execution is suspended after each yield until next() is called again

# Minimal Python book: Create functions

**Summary**

This [reading](https://paiml.com/docs/home/books/minimal-python/chapter03-create-functions) covers the basics of using functions in Python. It looks at functions with no inputs or returns, functions that return values without inputs, and functions that accept inputs but don't return anything. Key points are that functions allow reusable logic, can optionally take inputs and produce outputs, and handle the bulk of computation in most Python programs.

**Top 3 Key Points**

* Functions allow reusable blocks of logic in code
* Functions can optionally accept inputs and return outputs
* Functions form the core computation of most Python programs

# Generators

Summary:

[Generators](https://docs.python.org/3/tutorial/classes.html#generators) are functions that lazily produce a sequence of values using the yield keyword instead of returning values all at once like regular functions. This allows for lightweight lazy evaluation.

Key Points

* Generators use yield to output values one by one, suspending and resuming execution between each value
* Generators automatically save state between executions
* Anything you can do with generators can be done with class-based iterators, but generators are more compact
* Generator expressions provide a short syntax similar to list comprehensions

**Building Classes & Methods**

**Key Terms**

* **Class**: Blueprint for creating objects. Defines attributes and methods.
* **Instance**: Object that is created from a class. Has access to class attributes and methods.
* **Method**: Function that belongs to a class. Accessed via instance or class name.
* **Constructor**: Special method that runs when an instance is created. Used to initialize attributes.
* **Inheritance**: Creating a child class from parent class. Child inherits attributes and methods.

**Class Example**

class Vehicle:

    wheels = 4 # Class attribute

    def \_\_init\_\_(self, make, model):

        self.make = make # Instance attribute

        self.model = model

    def description(self): # Method

        return f"The {self.make} {self.model}"

car = Vehicle("BMW", "i3") # Instance created

print(car.wheels) # Access class attribute

print(car.description()) # Call instance method

4

The BMW i3

**Inheritance Example**

class Pet:

    def eat(self):

        print("Chomp")

class Dog(Pet):

    def bark(self):

        print("Bark!")

dog = Dog()

dog.eat() # Inherited method

dog.bark() # Dog specific method

Chomp

Bark!

**Class:** Defines a template

# Class

class UFC:

    pass

**Object:** An instance of a class

# Class

class UFC:

    pass

# Object

ufc = UFC()

**Attribute:** Variables bound to an object

class Fighter:

    def \_\_init\_\_(self):

        self.name = "Conor McGregor"  # Attribute

fighter = Fighter()

print(fighter.name)

Conor McGregor

**Method:** Functions defined in a class

class Fighter:

    def trash\_talk(self):# Method

        print("I'm the best!")

fighter = Fighter()

fighter.trash\_talk()

I'm the best!

**Inheritance:** Child class inherits from parent class

class Athlete:

    pass

class Fighter(Athlete):# Inheritance

    pass

**Python Code Example: Simple Inheritance**

# Create a simple Competitor class

class Competitor:

  def \_\_init\_\_(self, name, age, weight):

    self.name = name

    self.age = age

    self.weight = weight

 # Method Prints competitors stats

  def print\_stats(self):

    print(f"{self.name} is {self.age} years old and weighs {self.weight} pounds.")

# Create a Fighter class that inherits from Competitor

class Fighter(Competitor):

  pass

fighter = Fighter("Conor McGregor", 32, 170)

fighter.print\_stats()

Conor McGregor is 32 years old and weighs 170 pounds.

**Python Code Example: Using Inheritance**

# Example using inheritance

class UFC:

  def weight\_class(self, weight):

      # Maps weight to weight class

      return "Lightweight"

# Fighter class inherits from UFC

class Fighter(UFC):

  def \_\_init\_\_(self, name):

    self.name = name

  def print\_name(self):

    print(self.name)

fighter = Fighter("Khabib")

print(fighter.weight\_class(155))

fighter.print\_name()

Lightweight

Khabib

**Inheritance and OOP Fundamentals Reading**

This reading covers key concepts related to inheritance and object-oriented programming in Python. [Full reading here](https://docs.python.org/3/tutorial/classes.html#inheritance).

**Key Points**

* Derived classes inherit attributes and behaviors from parent base classes
* Overriding inherited methods allows modifying functionality in child classes
* Multiple inheritance allows a derived class to inherit from multiple parent classes
* Name mangling with double underscores enables a form of private class variables

**Summary of Lesson**

This lesson provided an introduction to using classes and methods in Python. We looked at defining classes and instantiating objects from them. We also covered special methods like constructors, adding regular methods, inheritance between classes, and accessing attributes and methods.

**Top 3 Key Points**

* Classes define attributes and behaviors for objects
* Instances are created from classes via initialization
* Methods operate on instance and class data

**Modules & Advanced Usage**

**Key Terms**

* **Module** - A Python file containing reusable code like functions or classes
* **Import** - Retrieves modules making their contents available in current namespace
* **Virtualenv** - Self-contained directory housing isolated Python packages/dependencies
* **Activation** - Configures shell to use virtualenv's dedicated Python interpreter
* **Pip** - Python tool for installing/managing packages and dependencies

# Import module

import utils

# Access function

print(utils.format\_name("Wyclef"))

**Using built-in Python virtualenv**

# Virtualenv creation

python3 -m venv env

# Activate virtualenv

source env/bin/activate

# Install dependency

pip install pandas

# Deactivate when done

Deactivate

# Understanding 3rd Party Packaging

This [template](https://github.com/nogibjj/mlops-template) provides a starting point for ML/DL projects leveraging GPU acceleration. It uses mainstream Python tools like virtualenv, pip, Docker, PyTorch, and TensorFlow to configure a development environment with GPU access, isolate dependencies, and test GPU training.

**Top 4 Key Points**

* Check virtualenv is active to manage packages separately
* Dockerfiles included to build GPU container images
* PyTorch and TensorFlow tests validate GPU works
* Tools like BentoML and Hugging Face integrate nicely