

**Repetition**

# Loops

- The foo-bang problem
  - Print a list of all numbers between 1 and 100
    - Whenever the number is divisible by 3, print "foo" afterwards
    - Whenever the number is divisible by 5, print "bang"
    - If both conditions are set, print "foo-bang"
- This is a version of a well-known easy programming interview question

# Loops

```
def foo_bang(n):  
    for i in range(1, n+1):  
        if i%3 == 0 and i%5 == 0:  
            print(i, 'foobang')  
        elif i%3 == 0 and i%5 != 0:  
            print(i, 'foo')  
        elif i%3 != 0 and i%5 == 0:  
            print(i, 'bang')  
        elif i%3 != 0 and i%5 != 0:  
            print(i)  
        else:  
            print(f'With {i}: Logic error in program')
```

# Loops

- A use case for continue
  - Find all numbers  $x$  between 1 and 10000 that satisfy the following conditions simultaneously:
    - $x$  is not a square
    - $x^3 + 2$  is divisible by 4
    - $(x - 1)(x + 1)$  is not divisible by 7
    - $(x + 2)(x + 3)$  is divisible by 5

# Loops

- Find the smallest number  $n$  such that  $\sum_{i=1}^n \frac{1}{i^2} > \sqrt{2} + \frac{1}{10}$

# Loops

- Each letter stands for a different digit.
- Solve:

- $$\begin{array}{r} \text{SEND} \\ +\text{MORE} \\ \hline \text{MONEY} \end{array}$$

# Loops

- Each letter stands for a different digit
  - Solve
  -

$$\begin{array}{r} \text{six} \\ + \text{six} \\ + \text{six} \\ = \\ \text{nine} \\ + \text{nine} \end{array}$$

# Loops

- Solve:

- 

```
    nine
  +seven
  +seven
  +seven
  -----
  thirty
```



# Classes and Object 1

Thomas Schwarz, SJ  
Marquette University

# Classes and Objects

- Imperative programming manipulates the state of memory
  - Breaks up tasks through procedures and functions
- Object Oriented Programming
  - Creates objects that interact with each other
  - Objects are defined with a user-defined data type

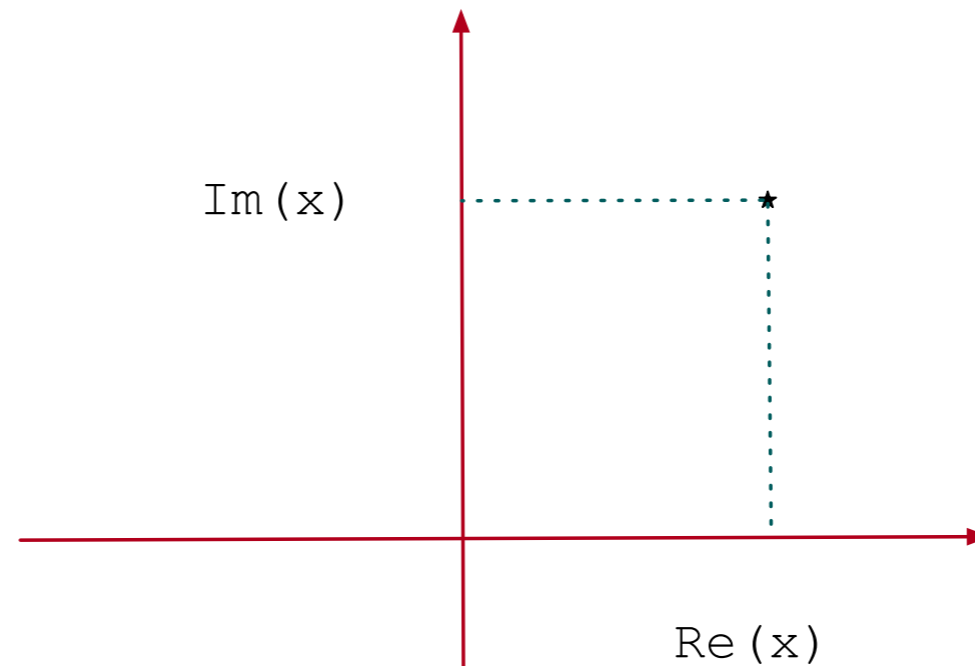
# Classes and Objects

- Each object maintains its own state
  - Objects manipulate themselves and other objects through methods

# Classes and Objects

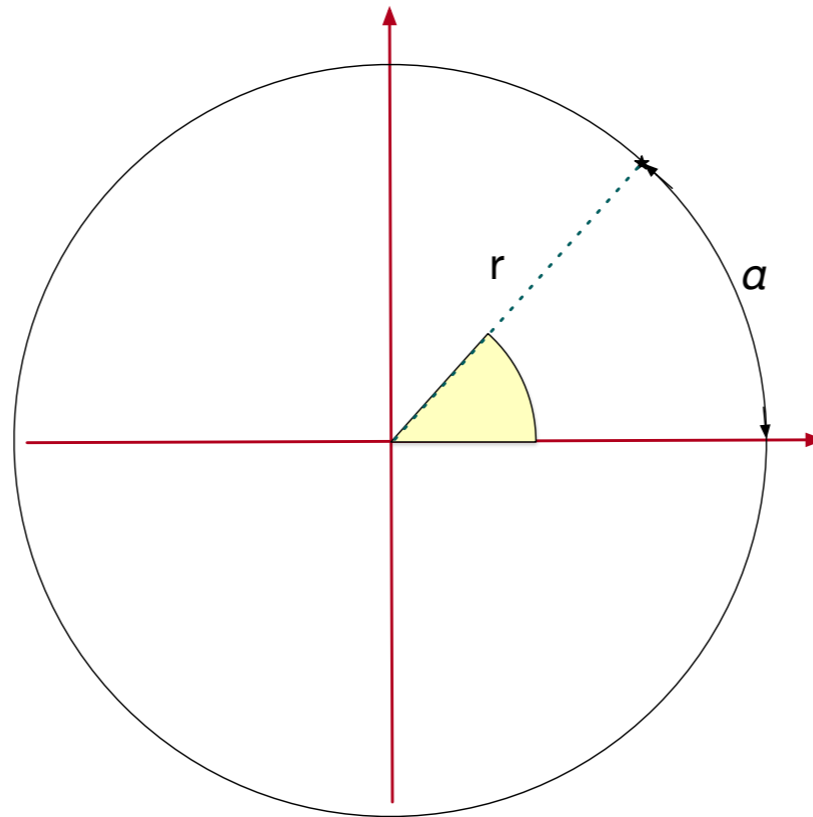
- Running Example:
  - Complex Numbers
  - Complex numbers live in the complex plane
    - Two canonical way of representing them:
      - Via coordinates (the real and the imaginary part)
      - Via length and angle to x-axis

# Complex Numbers



- Complex numbers are points in the Gaussian plane
- Can be represented as pairs  $(a,b)$ 
  - $a$  is called the real part,  $b$  is called the imaginary part
  - Written as  $a+ib$ , the “algebraic notation”

# Complex Numbers



$r$  is the length  
 $\alpha$  is the phase in  
radians

- Polar form
  - Given by angle  $\alpha$  with x-axis and a length  $r$
  - Written as  $r \cdot e^{i\alpha}$



# Complex Numbers

- Complex numbers allow various operations such as addition, multiplication, exponentiation,
- They have a length, a real part, and imaginary part, and a phase
- They have a transpose
- And so much more

# OOP in Python

- We define the type of an object as a class
  - Objects have fields and methods
    - Fields are like variables
    - Methods are like functions
- A complex number has two fields: the real and the imaginary part
- A method would be the calculation of the length
- Another standard method would be a string describing the number



# OOP in Python

- There are two types of fields and methods:
  - Those that belong to the class:
    - Class variables (aka Class fields), Class methods
  - And those that belong to an object
    - Object variables, Object methods

# OOP in Python

- To create an object of type class, “**instantiation**”: we define and use an initializer called `__init__()`
- The initializer can have arguments.
- If we create object variables and methods, we use the (quasi-)keyword `self` to refer to the object.

# OOP in Python

The double underscore before and after make this a reserved method name.

```
class Complex:
    def __init__(self, real, imaginary):
        self.re = real
        self.im = imaginary
    def __str__(self):
        return "{}+i{}".format(self.re, self.im)

if __name__ == "__main__":
    a = Complex(-2, 3)
    b = Complex(1, 2)
    print(a, b)
```

They are called dunder methods.

# OOP in Python

The “self” is required. It renders this is an instance method

```
class Complex():
    def __init__(self, real, imaginary):
        self.re = real
        self.im = imaginary
    def __str__(self):
        return "{}+i{}".format(self.re, self.im)

if __name__ == "__main__":
    a = Complex(-2, 3)
    b = Complex(1, 2)
    print(a, b)
```

# OOP in Python

The real and imaginary are parameters

```
class Complex():
    def __init__(self, real, imaginary):
        self.re = real
        self.im = imaginary
    def __str__(self):
        return "{}+i{}".format(self.re, self.im)

if __name__ == "__main__":
    a = Complex(-2, 3)
    b = Complex(1, 2)
    print(a, b)
```

# OOP in Python

```
class Complex():
    def __init__(self, real, imaginary):
        self.re = real
        self.im = imaginary
    def __str__(self):
        return "{}+i{}".format(self.re, self.im)

if __name__ == "__main__":
    a = Complex(-2, 3)
    b = Complex(1, 2)
    print(a, b)
```

**This is the creation of an instance of the class. "self" is hidden, -2 is real and 3 is imaginary**

# OOP in Python

This defines an instance field called "re"

```
class Complex():
    def __init__(self, real, imaginary):
        self.re = real
        self.im = imaginary
    def __str__(self):
        return "{}+i{}".format(self.re, self.im)

if __name__ == "__main__":
    a = Complex(-2, 3)
    b = Complex(1, 2)
    print(a, b)
```

# OOP in Python

This defines an instance field called "im"

```
class Complex():
    def __init__(self, real, imaginary):
        self.re = real
        self.im = imaginary
    def __str__(self):
        return "{}+i{}".format(self.re, self.im)

if __name__ == "__main__":
    a = Complex(-2, 3)
    b = Complex(1, 2)
    print(a, b)
```



# OOP in Python

Assigning self.whatever anywhere in the definition of the class will create an instance object

```
class Complex():
    def __init__(self, real, imaginary):
        self.re = real
        self.im = imaginary
    def __str__(self):
        return "{}+i{}".format(self.re, self.im)

if __name__ == "__main__":
    a = Complex(-2, 3)
    b = Complex(1, 2)
    print(a, b)
```

# OOP in Python

The two underscores before and after denotes `__str__` as a reserved name.

```
class Complex():
    def __init__(self, real, imaginary):
        self.re = real
        self.im = imaginary
    def __str__(self):
        return "{}+i{}".format(self.re, self.im)

if __name__ == "__main__":
    a = Complex(-2, 3)
    b = Complex(1, 2)
    print(a, b)
```

# OOP in Python

**`__str__` takes the instance and creates a string. The string should reflect the contents of the object.**

```
class Complex():
    def __init__(self, real, imaginary):
        self.re = real
        self.im = imaginary
    def __str__(self):
        return "{}+i{}".format(self.re, self.im)

if __name__ == "__main__":
    a = Complex(-2, 3)
    b = Complex(1, 2)
    print(a, b)
```

# OOP in Python

When `print` is called on objects of type `complex`, then Python looks first for a `__str__` method and then for a `__repr__` method. The `__repr__` method is supposed to give more details for debugging.

```
class Complex():
    def __init__(self, real, imaginary):
        self.re = real
        self.im = imaginary
    def __str__(self):
        return "{}+i{}".format(self.re, self.im)

if __name__ == "__main__":
    a = Complex(-2, 3)
    b = Complex(1, 2)
    print(a, b)
```

# OOP in Python

- Adding Methods
  - Every complex number has a length:

$$|a + bi| = \sqrt{a^2 + b^2}$$

- To create a method calculating the length:
  - We need one argument: the object (the complex number) itself
  - This argument is called `self`

```
class Complex():
    def __init__(self, real, imaginary):
        self.re = real
        self.im = imaginary
    def __str__(self):
        if self.re==0 and self.im==0:
            return "0"
        if self.re==0 and self.im>0:
            return "i{}".format(self.im)
        if self.re==0 and self.im<0:
            return "-i{}".format(-self.im)
        if self.im<0:
            return "{}-i{}".format(self.re, -self.im)
        if self.im>0:
            return "{}+i{}".format(self.re, self.im)
        return str(self.re)
def length(self):
    return math.sqrt(self.re*self.re+self.im*self.im)

if __name__ == "__main__":
    a = Complex(-2, 3)
    print(a.length())
```

The one argument is self

```
class Complex():
    def __init__(self, real, imaginary):
        self.re = real
        self.im = imaginary
    def __str__(self):
        if self.re==0 and self.im==0:
            return "0"
        if self.re==0 and self.im>0:
            return "i{}".format(self.im)
        if self.re==0 and self.im<0:
            return "-i{}".format(-self.im)
        if self.im<0:
            return "{}-i{}".format(self.re, -self.im)
        if self.im>0:
            return "{}+i{}".format(self.re, self.im)
        return str(self.re)
def length(self):
    return math.sqrt(self.re*self.re+self.im*self.im)
```

```
if __name__ == "__main__":
    a = Complex(-2, 3)
    print(a.length())
```

**Here is how this function is called: The object is followed by a period**

```

class Complex():
    def __init__(self, real, imaginary):
        self.re = real
        self.im = imaginary
    def __str__(self):
        if self.re==0 and self.im==0:
            return "0"
        if self.re==0 and self.im>0:
            return "i{}".format(self.im)
        if self.re==0 and self.im<0:
            return "-i{}".format(-self.im)
        if self.im<0:
            return "{}-i{}".format(self.re, -self.im)
        if self.im>0:
            return "{}+i{}".format(self.re, self.im)
        return str(self.re)
def length(self):
    return math.sqrt(self.re*self.re+self.im*self.im)

```

```

if __name__ == "__main__":
    a = Complex(-2, 3)
    print(a.length())

```

**So, this method is really a function of one argument, even if it does not look like it.**



```

class Complex():
    def __init__(self, real, imaginary):
        self.re = real
        self.im = imaginary
    def __str__(self):
        if self.re==0 and self.im==0:
            return "0"
        if self.re==0 and self.im>0:
            return "i{}".format(self.im)
        if self.re==0 and self.im<0:
            return "-i{}".format(-self.im)
        if self.im<0:
            return "{}-i{}".format(self.re, -self.im)
        if self.im>0:
            return "{}+i{}".format(self.re, self.im)
        return str(self.re)
def length(self):
    return math.sqrt(self.re*self.re+self.im*self.im)

if __name__ == "__main__":
    a = Complex(-2, 3)
    print(a.length())

```

**Here we are referring to an instance field.**

# Self Test:

Stop the presentation and fire up IDLE

- The phase of a complex number is defined by

$$\mathbf{arg}(a + bi) = \begin{cases} \arctan\left(\frac{b}{a}\right) & \mathbf{if } a > 0 \\ \arctan\left(\frac{b}{a}\right) + \pi & \mathbf{if } a < 0 \mathbf{ and } b \geq 0 \\ \arctan\left(\frac{b}{a}\right) - \pi & \mathbf{if } a < 0 \mathbf{ and } b < 0 \\ \frac{\pi}{2} & \mathbf{if } a = 0 \mathbf{ and } b > 0 \\ -\frac{\pi}{2} & \mathbf{if } a = 0 \mathbf{ and } b < 0 \\ \mathbf{undefined} & \mathbf{if } x = 0 \mathbf{ and } b = 0 \end{cases}$$

- Raise a ValueError in the last case, the arctan in the math library is called atan.

```
class Complex():
    def arg(self):
        if self.re == 0 and self.im == 0:
            raise ValueError
        if self.re > 0:
            return math.atan(self.im/self.re)
        if self.re < 0 and self.im >= 0:
            return math.atan(self.im/self.re)+math.pi
        if self.re < 0 and self.im < 0:
            return math.atan(self.im/self.re) - math.pi
        if self.re == 0 and self.im > 0:
            return math.pi/2
        if self.re == 0 and self.im < 0:
            return -math.pi/2

if __name__ == "__main__":
    a = Complex(0, 0)
    print(a.arg())
```

# OOP in Python

- Methods can of course return other objects
  - The conjugate of a complex number is obtained by reflecting around the y-axis

$$\overline{a + bi} = a - bi$$

# OOP in Python

```
class Complex():  
    def conjugate(self):  
        return Complex(self.re, -self.im)
```

```
if __name__ == "__main__":  
    a = Complex(2, 3)  
    print(a.conjugate())
```

# OOP

- Python knows “operator overloading”
  - Instead of adding two complex numbers by saying something like `c=add(a,b)`
  - We can say `c=a+b`
- Python knows a number of such overloads and associates them by reserving function names

```
class Complex():
    def __add__(self, other):
        return Complex(self.re+other.re, self.im+other.im)

if __name__ == "__main__":
    a = Complex(2, 3)
    b = Complex(3, -5)
    print(a+b)
```

**We need to return a complex number**

**Here we are adding two complex numbers.**

# Self Test

- Overload the subtraction
  - The reserved method name is `__sub__`
    - (Two underscores before and after)



# Solution

```
class Complex():
    def __sub__(self, other):
        return Complex(self.re-other.re, self.im-other.im)

if __name__ == "__main__":
    a = Complex(2, 3)
    b = Complex(3, -5)
    suma = a+b
    dif = a-b
    print(a, b, suma, dif)
```

# OOP in Python

- We can define comparisons via overloading
  - For equality, the reserved method name is `__eq__`

# OOP in Python

- Resolving operators
  - Operator overloading does not have to happen between objects of the same type
  - If there is an expression `a==b`
    - Python first looks into the class definition of `a` for an `__eq__` method that has second parameter the type of `b`
    - If that fails, Python then looks into the class definition of `b` for an `__eq__` method

# OOP

- In fact, there is a default comparison between two objects
  - It is based on *IDENTITY NOT EQUALITY*
  - Identity: Two objects are stored at the same location
  - Equality: Two objects have the same fields

```
class Complex():
    def __eq__(self, other):
        if isinstance(other, Complex):
            return self.re==other.re and self.im==other.im
        return NotImplemented
    def __ne__(self, other):
        if isinstance(other, Complex):
            return self.re!=other.re or self.im!=other.im
        return NotImplemented

if __name__ == "__main__":
    a = Complex(2, 3)
    b = Complex(3, -5)
    print(a==b, a!=b)
    print(a==Complex(2,3), a!=Complex(2,3))
```

**We only want to compare two complex numbers**

```
class Complex():
    def __eq__(self, other):
        if isinstance(other, Complex):
            return self.re==other.re and self.im==other.im
        return NotImplemented
    def __ne__(self, other):
        if isinstance(other, Complex):
            return self.re!=other.re or self.im!=other.im
        return NotImplemented

if __name__ == "__main__":
    a = Complex(2, 3)
    b = Complex(3, -5)
    print(a==b, a!=b)
    print(a==Complex(2,3), a!=Complex(2,3))
```

**Two complex numbers are equal if they have the same real and imaginary parts**

```
class Complex():
    def __eq__(self, other):
        if isinstance(other, Complex):
            return self.re==other.re and self.im==other.im
        return NotImplemented
    def __ne__(self, other):
        if isinstance(other, Complex):
            return self.re!=other.re or self.im!=other.im
        return NotImplemented

if __name__ == "__main__":
    a = Complex(2, 3)
    b = Complex(3, -5)
    print(a==b, a!=b)
    print(a==Complex(2, 3), a!=Complex(2, 3))
```

**If we compare a complex number with a non-complex number then we want to return the constant `NotImplemented`**

# Classes and Objects

- Classes usually define objects, but they can also be used in isolation
  - Assume that you want to use a number of global variables
    - This is dangerous, since you might be reusing the same name
- Solution: Use a class that contains all these variables



# A Globals Class

- We call the class `G1` — short for global
- Store constants as class variables
- Easy to identify in program

```
class G1:
    gr2gr = 0.06479891
    dr2gr = 1.7718451953125
    oz2gr = 28.349523125
    lb2gr = 453.59237
    st2gr = 6350.29318
```

```
def translate(number, measure):
    if measure == "gr":
        return "{0:.3f} {1}".format(number*G1.gr2gr, "gram")
    if measure == "dr":
        return "{0:.3f} {1}".format(number*G1.dr2gr, "gram")
    if measure == "oz":
        return "{0:.3f} {1}".format(number*G1.oz2gr, "gram")
    if measure == "lb":
        return "{0:.3f} {1}".format(number*G1.lb2gr, "gram")
    if measure == "st":
        return "{0:.3f} {1}".format(number*G1.st2gr/1000, "kg")
    raise ValueError
```

# Class and Instance Variables

- Class variable
  - belong to the class
  - shared by all objects
  - defined without prefix in the class
- Instance variable
  - belong to the instance
  - not shared by objects
  - defined by using an object or self prefix

# Self Test

- Identify the type of the bold-faced variables in the following code

```
import math

class Example:
    exists = False
    def __init__(self, x, y):
        self.radius = math.sqrt(x*x+y+y)
        self.x = x
        self.y = y
        Example.exists = True

print(Example.exists)
e = Example(2, 3)
print(e.x)
print(Example.exists)
print(e.radius)
```

# Answer

```
import math
```

```
class Example:
```

```
    exists = False
```

```
    def __init__(self, x, y):
```

```
        self.radius = math.sqrt(x*x+y+y)
```

```
        self.x = x
```

```
        self.y = y
```

```
    Example.exists = True
```

```
print(Example.exists)
```

```
e = Example(2, 3)
```

```
print(e.x)
```

```
print(Example.exists)
```

```
print(e.radius)
```

This is an instance variable. It belongs to the (one and only) object of type Example.

It happens to be defined in `__init__`. However, it is defined with the `self` prefix.

# Answer

```
import math

class Example:
    exists = False
    def __init__(self, x, y):
        self.radius = math.sqrt(x*x+y+y)
        self.x = x
        self.y = y
        Example.exists = True

print(Example.exists)
e = Example(2, 3)
print(e.x)
print(Example.exists)
print(e.radius)
```

This is a class variable. It is specified by using the class name "Example."

It is defined without a prefix within the class.

# Answer

```
import math

class Example:
    exists = False
    def __init__(self, x, y):
        self.radius = math.sqrt(x*x+y+y)
        self.x = x
        self.y = y
        Example.exists = True

print(Example.exists)
e = Example(2, 3)
print(e.x)
print(Example.exists)
print(e.radius)
```

This is an instance variable. It is defined with the prefix self.

It is used by referring to an object e.

# Class and Instance Methods

- The same distinction can be made for methods
  - Methods are functions related to an object
- A class method depends only on the class.
  - It is defined in the class, but has no argument `self`
  - It is called by giving the class-name
- An instance method depends on an instance
  - It is defined in the class with first argument `self`
  - It is called by prefacing it with an instance.
    - The instance is called the implicit argument

# Class and Instance Methods

```
class Example:  
    def foo():  
        print("foo")  
    def __init__(self):  
        pass  
    def bar(self):  
        print("bar")
```

A method definition  
without argument self:  
Class Method

It is called using the  
class-name to call it

```
Example.foo()  
e = Example()  
e.bar()
```



# Class and Instance

## Methods

```
class Example:  
    def foo():  
        print("foo")  
    def __init__(self):  
        pass  
    def bar(self):  
        print("bar")
```

```
Example.foo()  
e = Example()  
e.bar()
```

A method definition with  
argument self:  
Instance Method

It is called using the  
Instance.  
Without an object e, we  
cannot call it.

# Self Test

- Identify the type of methods in the following code

```
import math
class Vector3D:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z
    def zeroes():
        return Vector3D(0,0,0)
    def ones():
        return Vector3D(1,1,1)
    def __add__(self, other):
        return Vector3D(self.x+other.x,
                        self.y+other.y,
                        self.z+other.z)
    def __str__(self):
        return "({}, {}, {})".format(self.x, self.y, self.z)
    def length(self):
        return math.sqrt(self.x**2+self.y**2+self.z**2)
```

# Answers

```
import math
class Vector3D:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z
    def zeroes():
        return Vector3D(0,0,0)
    def ones():
        return Vector3D(1,1,1)
    def __add__(self, other):
        return Vector3D(self.x+other.x,
                        self.y+other.y,
                        self.z+other.z)
    def __str__(self):
        return "({}, {}, {})".format(self.x, self.y, self.z)
    def length(self):
        return math.sqrt(self.x**2+self.y**2+self.z**2)
```

**Dunder (double under) method:  
Hard to tell**

# Answers

```
import math
class Vector3D:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z
    def zeroes():
        return Vector3D(0,0,0)
    def ones():
        return Vector3D(1,1,1)
    def __add__(self, other):
        return Vector3D(self.x+other.x,
                        self.y+other.y,
                        self.z+other.z)
    def __str__(self):
        return "({}, {}, {})".format(self.x, self.y, self.z)
    def length(self):
        return math.sqrt(self.x**2+self.y**2+self.z**2)
```

**Class Method, even though it generates an object**

# Answers

```
import math
class Vector3D:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z
    def zeroes():
        return Vector3D(0,0,0)
    def ones():
        return Vector3D(1,1,1)
    def __add__(self, other):
        return Vector3D(self.x+other.x,
                        self.y+other.y,
                        self.z+other.z)
    def __str__(self):
        return "({}, {}, {})".format(self.x, self.y, self.z)
    def length(self):
        return math.sqrt(self.x**2+self.y**2+self.z**2)
```

**Class Method, even though it generates an object**

# Answers

```
import math
class Vector3D:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z
    def zeroes():
        return Vector3D(0,0,0)
    def ones():
        return Vector3D(1,1,1)
    def __add__(self, other):
        return Vector3D(self.x+other.x,
                        self.y+other.y,
                        self.z+other.z)
    def __str__(self):
        return "({}, {}, {})".format(self.x, self.y, self.z)
    def length(self):
        return math.sqrt(self.x**2+self.y**2+self.z**2)
```

**Instance method**

# Answers

```
import math
class Vector3D:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z
    def zeroes():
        return Vector3D(0,0,0)
    def ones():
        return Vector3D(1,1,1)
    def __add__(self, other):
        return Vector3D(self.x+other.x,
                        self.y+other.y,
                        self.z+other.z)
    def __str__(self):
        return "({}, {}, {})".format(self.x, self.y, self.z)
    def length(self):
        return math.sqrt(self.x**2+self.y**2+self.z**2)
```

**Dunder instance method**

# Answers

```
import math
class Vector3D:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z
    def zeroes():
        return Vector3D(0,0,0)
    def ones():
        return Vector3D(1,1,1)
    def __add__(self, other):
        return Vector3D(self.x+other.x,
                        self.y+other.y,
                        self.z+other.z)
    def __str__(self):
        return "({}, {}, {})".format(self.x, self.y, self.z)
    def length(self):
        return math.sqrt(self.x**2+self.y**2+self.z**2)
```

**Instance method**



# Dunder Methods

- Python reserves special names for functions that allows the programmer to emulate the behavior of built-in types
  - For example, we can create number like objects that allow for operations such as addition and multiplication
  - These methods have special names that start out with two underscores and end with two underscores
- Aside: If you preface a variable / function / class with a single underscore, you indicate that it should be treated as reserved and not used outside of the module / class

# Dunder Method

- A class for playing cards:
  - A card has a suit and a rank
    - We define this in the constructor `__init__`

```
class Card:  
    def __init__(self, suit, rank):  
        self.suit = suit  
        self.rank = rank
```

# Dunder Method

- We want to print it
  - Python likes to have two methods:
    - `__repr__` for more information, e.g. errors
    - `__str__` for the print-function
    - Both return a string

```
class Card:
```

```
    def __str__(self):  
        return self.suit[0:2]+self.rank[0:2]  
    def __repr__(self):  
        return "{}-{}".format(self.suit, self.rank)
```

# Dunder Method

- `__repr__` is used when we create an object in the terminal

```
>>> Card("Heart", "Queen")  
Heart-Queen
```

- `__str__` is used within `print` or when we say `str(card)`

```
>>> print(Card("Heart", "Queen"))  
HeQu  
>>> str(Card("Heart", "Queen"))  
'HeQu'
```

# Dunder Method

- We now create a carddeck class
  - Consists of a set of cards
  - Constructor uses a list of ranks and a list of suits

```
class Deck:  
    def __init__(self, los, lov):  
        self.cards = [Card(suit, rank) for suit in los  
                      for rank in lov]
```

# Dunder Method

- We create the string method. Remember that it needs to return a string.

```
class Deck:
    def __init__(self, los, lov):
        self.cards = [Card(suit, rank) for suit in los
                      for rank in lov]

    def __str__(self):
        result = []
        for card in self.cards:
            result.append(str(card))
        return " ".join(result)
```

# Dunder Method

- In order to allow python to check whether a deck exists, we want to have a length class. Besides, it is useful in itself.
- `if deck:` works by checking `len(deck)`

```
class Deck:  
  
    def __len__(self):  
        return len(self.cards)
```

# Dunder Method

- Given a deck, we want to be able to access the i-th element.
- We do so by defining `__getitem__`

```
class Deck:  
  
    def __getitem__(self, position):  
        return self.cards[position]
```



# Dunder Method

- This turns out to be very powerful:

```
french_deck = Deck(['Spade', 'Diamonds', 'Hearts', 'Clubs'],  
                  ['Ace', 'King', 'Queen', 'Jack', '10', '9',  
                  '8', '7', '6', '5', '4', '3', '2'])
```

- We can print out the *i*-th element of the deck

```
>>> str(french_deck[5])  
'Sp9'
```

- But we can also **slice** the deck

```
>>> print(french_deck[6:12])  
[Spade-8, Spade-7, Spade-6, Spade-5, Spade-4, Spade-3]
```

# Dunder Method

- We can use `random.choice()` to select a card

```
>>> random.choice(french_deck)
Diamonds-9
```

- Only for `random.sample` do we need to go to the underlying instance field

```
>>> random.sample(french_deck.cards, 5)
[Hearts-8, Hearts-2, Hearts-Ace, Hearts-6, Diamonds-Ace]
>>> random.sample(french_deck.cards, 5)
[Hearts-5, Clubs-Queen, Diamonds-Ace, Clubs-3, Clubs-King]
```

- But this is ugly and we better write a class method for it.