Solutions

For the remaining skeptics ...

Instantiation	>>> a = Polygon("Polly") (Creating an instance of the class Polygon)	>>> b = “Polygon”

Types	>>> type(a)
	<type 'instance'>
	>>> type(type(a))
	<type 'type'>
		>>> type(b)
		<type 'str'>
		>>> type(type(b))
		<type 'type'>

Methods	>>> a.print_name()
	Hi, my name is Polly.
	>>> a.perimeter()
	0
		>>> b.upper()
		POLYGON
		>>> b.replace(“gon”, “wog”)
		Polywog

Because of the way Python is set up, you have been using object-oriented techniques this entire time!

A “Procedural” Approach

```
>>> import math
>>> def perimeter(polygon):
...    """Given a list of vector vertices (in proper order), returns
...       the perimeter for the associated polygon."""
...    sum = 0
...    for i in range(len(polygon)):
...       vertex1 = polygon[i]
...       vertex2 = polygon[(i+1) % len(polygon)]
...       distance = math.sqrt(pow(vertex2[0]-vertex1[0],2) + \
...                            pow(vertex2[1]-vertex1[1],2))
...       sum += distance
...    return sum
...
>>> perimeter([[0,0],[1,0],[1,1],[0,1]])
4.0
>>> perimeter([[0,-2],[1,1],[3,3],[5,1],[4,0],[4,-3]])
17.356451097651515
```

Define a function that takes vertices as input (i.e., variable) and returns perimeter

An Object-Oriented Approach

```
>>> import math
>>> class Polygon:
...    “””A new class named Polygon.”””
...    def __init__(self, vertices=[]):
...       self.vertices = vertices
...       print "(Creating an instance of the class Polygon)"
...    def perimeter(self):
...       sum = 0
...       for i in range(len(self.vertices)):
...          vertex1 = self.vertices[i]
...          vertex2 = self.vertices[(i+1) % len(self.vertices)]
...          distance = math.sqrt(pow(vertex2[0]-vertex1[0],2) + \
...                               pow(vertex2[1]-vertex1[1],2))
...          sum += distance
...       return sum
...
>>> a = Polygon([[0,-2],[1,1],[3,3],[5,1],[4,0],[4,-3]])
>>> a.perimeter()
17.356451097651515
```

A “Procedural” Approach

```
>>> import math
>>> def perimeter(polygon):
...     """Given a list of vector vertices (in proper order), returns
...        the perimeter for the associated polygon."""
...     sum = 0
...     for i in range(len(polygon)):
...        vertex1 = polygon[i]
...        vertex2 = polygon[(i+1) % len(polygon)]
...        distance = math.sqrt(pow(vertex2[0]-vertex1[0],2) + \
...                             pow(vertex2[1]-vertex1[1],2))
...        sum += distance
...     return sum
...
>>> perimeter([[0,0],[1,0],[1,1],[0,1]])
4.0
>>> perimeter([[0,-2],[1,1],[3,3],[5,1],[4,0],[4,-3]])
17.356451097651515
```

Imports math module and associated routines

A “Procedural” Approach

```
>>> import math
>>> def perimeter(polygon):
...    """Given a list of vector vertices (in proper order), returns
...       the perimeter for the associated polygon."""
...    sum = 0
...    for i in range(len(polygon)):
...       vertex1 = polygon[i]
...       vertex2 = polygon[(i+1) % len(polygon)]
...       distance = math.sqrt(pow(vertex2[0]-vertex1[0],2) + \
...                            pow(vertex2[1]-vertex1[1],2))
...       sum += distance
...    return sum
...
>>> perimeter([[0,0],[1,0],[1,1],[0,1]])
4.0
>>> perimeter([[0,-2],[1,1],[3,3],[5,1],[4,0],[4,-3]])
17.356451097651515
```

Define a new function named *perimeter* that requires a single argument named *polygon*

A “Procedural” Approach

```
>>> import math
>>> def perimeter(polygon):
...     """Given a list of vector vertices (in proper order), returns
...        the perimeter for the associated polygon."""
...     sum = 0
...     for i in range(len(polygon)):
...        vertex1 = polygon[i]
...        vertex2 = polygon[(i+1) % len(polygon)]
...        distance = math.sqrt(pow(vertex2[0]-vertex1[0],2) + \
...                             pow(vertex2[1]-vertex1[1],2))
...        sum += distance
...     return sum
...
>>> perimeter([[0,0],[1,0],[1,1],[0,1]])
4.0
>>> perimeter([[0,-2],[1,1],[3,3],[5,1],[4,0],[4,-3]])
17.356451097651515
```

A documentation string describes (in English) the purpose of the function

A “Procedural” Approach

```
>>> import math
>>> def perimeter(polygon):
...    """Given a list of vector vertices (in proper order), returns
...       the perimeter for the associated polygon."""
...    sum = 0
...    for i in range(len(polygon)):
...       vertex1 = polygon[i]
...       vertex2 = polygon[(i+1) % len(polygon)]
...       distance = math.sqrt(pow(vertex2[0]-vertex1[0],2) + \
...                            pow(vertex2[1]-vertex1[1],2))
...       sum += distance
...    return sum
...
>>> perimeter([[0,0],[1,0],[1,1],[0,1]])
4.0
>>> perimeter([[0,-2],[1,1],[3,3],[5,1],[4,0],[4,-3]])
17.356451097651515
```

Initializes the variable *sum*

A “Procedural” Approach

```
>>> import math
>>> def perimeter(polygon):
...     """Given a list of vector vertices (in proper order), returns
...        the perimeter for the associated polygon."""
...     sum = 0
...     for i in range(len(polygon)):
...        vertex1 = polygon[i]
...        vertex2 = polygon[(i+1) % len(polygon)]
...        distance = math.sqrt(pow(vertex2[0]-vertex1[0],2) + \
...                             pow(vertex2[1]-vertex1[1],2))
...        sum += distance
...     return sum
...
>>> perimeter([[0,0],[1,0],[1,1],[0,1]])
4.0
>>> perimeter([[0,-2],[1,1],[3,3],[5,1],[4,0],[4,-3]])
17.356451097651515
```

Loop over each individual vertex in the variable *polygon*

A “Procedural” Approach

```
>>> import math
>>> def perimeter(polygon):
...     """Given a list of vector vertices (in proper order), returns
...        the perimeter for the associated polygon."""
...     sum = 0
...     for i in range(len(polygon)):
...        vertex1 = polygon[i]
...        vertex2 = polygon[(i+1) % len(polygon)]
...        distance = math.sqrt(pow(vertex2[0]-vertex1[0],2) + \
...                             pow(vertex2[1]-vertex1[1],2))
...        sum += distance
...     return sum
...
>>> perimeter([[0,0],[1,0],[1,1],[0,1]])
4.0
>>> perimeter([[0,-2],[1,1],[3,3],[5,1],[4,0],[4,-3]])
17.356451097651515
```

Grab adjacent vertices. Modulo operator (%) avoids list index exception

A “Procedural” Approach

```
>>> import math
>>> def perimeter(polygon):
...    """Given a list of vector vertices (in proper order), returns
...       the perimeter for the associated polygon."""
...    sum = 0
...    for i in range(len(polygon)):
...       vertex1 = polygon[i]
...       vertex2 = polygon[(i+1) % len(polygon)]
...       distance = math.sqrt(pow(vertex2[0]-vertex1[0],2) + \
...                            pow(vertex2[1]-vertex1[1],2))
...       sum += distance
...    return sum
...
>>> perimeter([[0,0],[1,0],[1,1],[0,1]])
4.0
>>> perimeter([[0,-2],[1,1],[3,3],[5,1],[4,0],[4,-3]])
17.356451097651515
```

Calculate the distance between adjacent vertices

A “Procedural” Approach

```
>>> import math
>>> def perimeter(polygon):
...     """Given a list of vector vertices (in proper order), returns
...        the perimeter for the associated polygon."""
...     sum = 0
...     for i in range(len(polygon)):
...        vertex1 = polygon[i]
...        vertex2 = polygon[(i+1) % len(polygon)]
...        distance = math.sqrt(pow(vertex2[0]-vertex1[0],2) + \
...                             pow(vertex2[1]-vertex1[1],2))
...        sum += distance
...     return sum
...
>>> perimeter([[0,0],[1,0],[1,1],[0,1]])
4.0
>>> perimeter([[0,-2],[1,1],[3,3],[5,1],[4,0],[4,-3]])
17.356451097651515
```

Increment the *distance* variable

A “Procedural” Approach

```
>>> import math
>>> def perimeter(polygon):
...    """Given a list of vector vertices (in proper order), returns
...       the perimeter for the associated polygon."""
...    sum = 0
...    for i in range(len(polygon)):
...       vertex1 = polygon[i]
...       vertex2 = polygon[(i+1) % len(polygon)]
...       distance = math.sqrt(pow(vertex2[0]-vertex1[0],2) + \
...                            pow(vertex2[1]-vertex1[1],2))
...       sum += distance
...    return sum
...
>>> perimeter([[0,0],[1,0],[1,1],[0,1]])
4.0
>>> perimeter([[0,-2],[1,1],[3,3],[5,1],[4,0],[4,-3]])
17.356451097651515
```

Return the value of *sum*

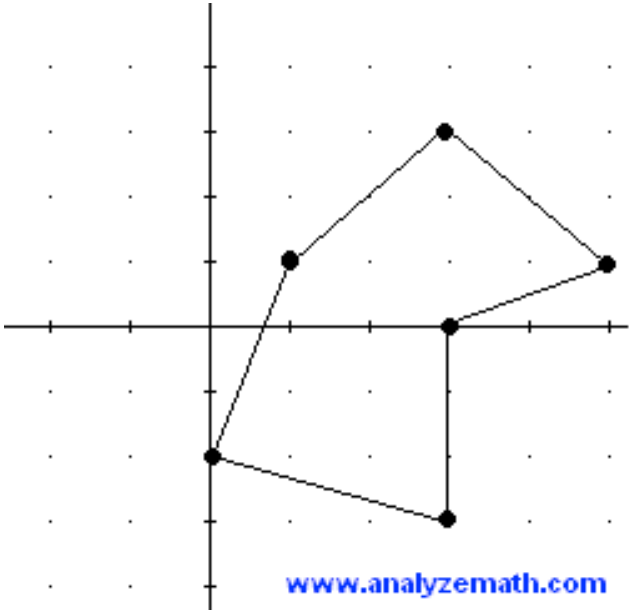
A “Procedural” Approach

```
>>> import math
>>> def perimeter(polygon):
...    """Given a list of vector vertices (in proper order), returns
...       the perimeter for the associated polygon."""
...    sum = 0
...    for i in range(len(polygon)):
...       vertex1 = polygon[i]
...       vertex2 = polygon[(i+1) % len(polygon)]
...       distance = math.sqrt(pow(vertex2[0]-vertex1[0],2) + \
...                            pow(vertex2[1]-vertex1[1],2))
...       sum += distance
...    return sum
...
>>> perimeter([[0,0],[1,0],[1,1],[0,1]])
4.0
>>> perimeter([[0,-2],[1,1],[3,3],[5,1],[4,0],[4,-3]])
17.356451097651515
```

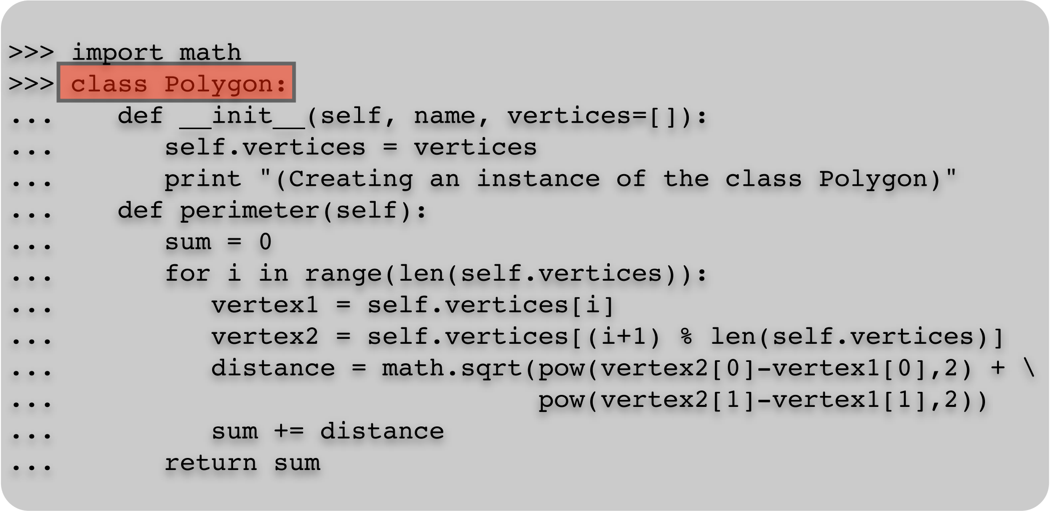
Unit square

A “Procedural” Approach

```
>>> import math
>>> def perimeter(polygon):
...     """Given a list of vector vertices (in proper order), returns
...        the perimeter for the associated polygon."""
...     sum = 0
...     for i in range(len(polygon)):
...        vertex1 = polygon[i]
...        vertex2 = polygon[(i+1) % len(polygon)]
...        distance = math.sqrt(pow(vertex2[0]-vertex1[0],2) + \
...                             pow(vertex2[1]-vertex1[1],2))
...        sum += distance
...     return sum
...
>>> perimeter([[0,0],[1,0],[1,1],[0,1]])
4.0
>>> perimeter([[0,-2],[1,1],[3,3],[5,1],[4,0],[4,-3]])
17.356451097651515
```



An Object-Oriented Approach



```
>>> import math
>>> class Polygon:
...    def __init__(self, name, vertices=[]):
...       self.vertices = vertices
...       print "(Creating an instance of the class Polygon)"
...    def perimeter(self):
...       sum = 0
...       for i in range(len(self.vertices)):
...          vertex1 = self.vertices[i]
...          vertex2 = self.vertices[(i+1) % len(self.vertices)]
...          distance = math.sqrt(pow(vertex2[0]-vertex1[0],2) + \
...                               pow(vertex2[1]-vertex1[1],2))
...          sum += distance
...       return sum
```

Define a new class named *polygon.* Note that the class name is not followed by parentheses (unless it has a parent class - more on this in the next lecture)

An Object-Oriented Approach

```
>>> import math
>>> class Polygon:
...     def __init__(self, name, vertices=[]):
...         self.vertices = vertices
...         print "(Creating an instance of the class Polygon)"
...     def perimeter(self):
...         sum = 0
...         for i in range(len(self.vertices)):
...             vertex1 = self.vertices[i]
...             vertex2 = self.vertices[(i+1) % len(self.vertices)]
...             distance = math.sqrt(pow(vertex2[0]-vertex1[0],2) + \
...                                  pow(vertex2[1]-vertex1[1],2))
...             sum += distance
...         return sum
```

__init__ is a special Python method. It is called every time a new instance of a class is created

An Object-Oriented Approach

```
>>> import math
>>> class Polygon:
...    def __init__(self, name, vertices=[]):
...       self.vertices = vertices
...       print "(Creating an instance of the class Polygon)"
...    def perimeter(self):
...       sum = 0
...       for i in range(len(self.vertices)):
...          vertex1 = self.vertices[i]
...          vertex2 = self.vertices[(i+1) % len(self.vertices)]
...          distance = math.sqrt(pow(vertex2[0]-vertex1[0],2) + \
...                               pow(vertex2[1]-vertex1[1],2))
...          sum += distance
...       return sum
```

self is also a special Python object. It is essentially a reference to the specific instance of the class

An Object-Oriented Approach

```
>>> import math
>>> class Polygon:
...    def __init__(self, name, vertices=[]):
...       self.vertices = vertices
...       print "(Creating an instance of the class Polygon)"
...    def perimeter(self):
...       sum = 0
...       for i in range(len(self.vertices)):
...          vertex1 = self.vertices[i]
...          vertex2 = self.vertices[(i+1) % len(self.vertices)]
...          distance = math.sqrt(pow(vertex2[0]-vertex1[0],2) + \
...                               pow(vertex2[1]-vertex1[1],2))
...          sum += distance
...       return sum
```

The arguments that follow *self* in the declaration of the *__init__* method (and all others, for that matter), are just like other Python arguments. In this case, *name* is required and *vertices* is optional

An Object-Oriented Approach

```
>>> import math
>>> class Polygon:
...    def __init__(self, name, vertices=[]):
...       self.vertices = vertices
...       print "(Creating an instance of the class Polygon)"
...    def perimeter(self):
...       sum = 0
...       for i in range(len(self.vertices)):
...          vertex1 = self.vertices[i]
...          vertex2 = self.vertices[(i+1) % len(self.vertices)]
...          distance = math.sqrt(pow(vertex2[0]-vertex1[0],2) + \
...                               pow(vertex2[1]-vertex1[1],2))
...          sum += distance
...       return sum
```

The initialization steps. Whenever a new instance of the *Polygon* class is created, the attribute *vertices* is set, and a message is printed to stdout