CS 134:
Classes and Objects

Announcements & Logistics

- **Lab 7** is today/tomorrow
  - Due Wed/Thurs, 11 pm
  - Complete **Task 0** before lab
  - Remember to **add, commit and push** code and **required images**
  - Quick note about **command line arguments** (in Lab 7)

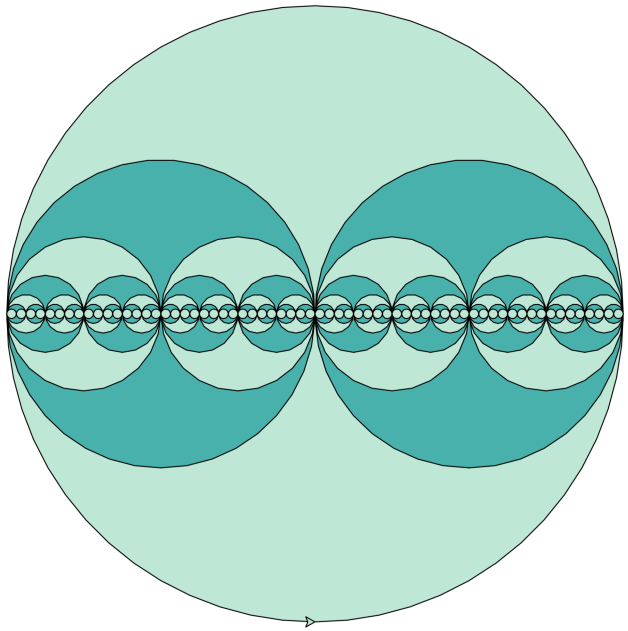
    ```
    >>> python3 bedtime.py duck cow dog
    ```

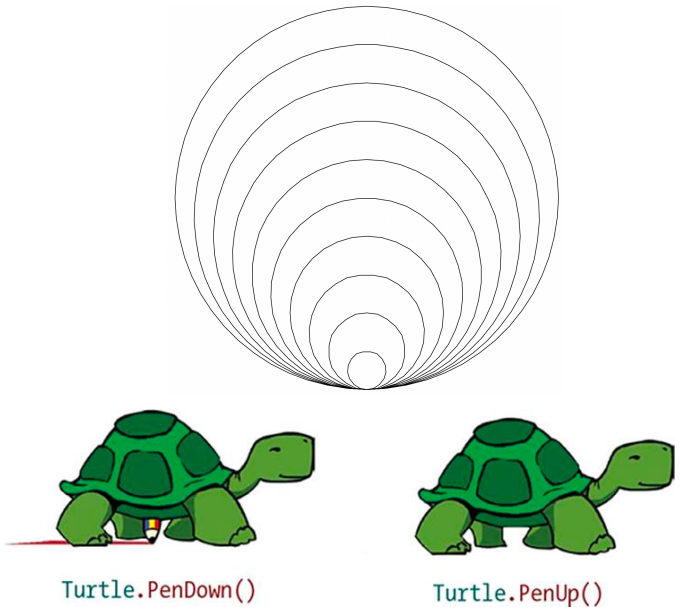
  - Interpreted as a list of strings; we provide the code for you
- **HW 6** due tonight at 11 pm (on Glow)

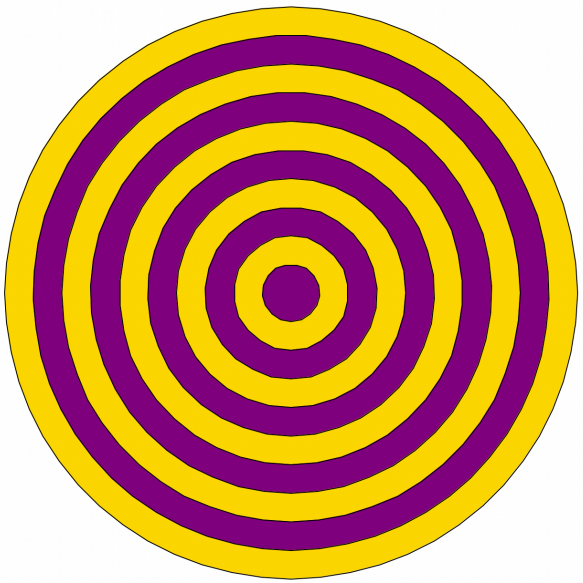
Do You Have Any Questions?

Last Time

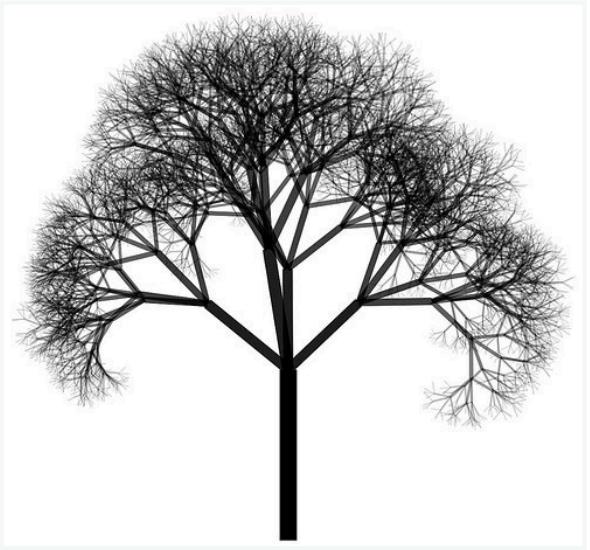
- Learned about the Python Turtle package
- Investigated graphical recursion examples
- Learned about function **invariance** and why it really matters when doing recursion







Today

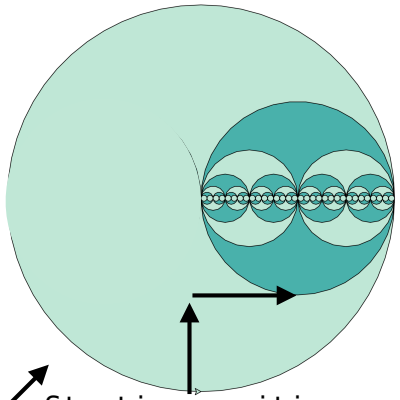


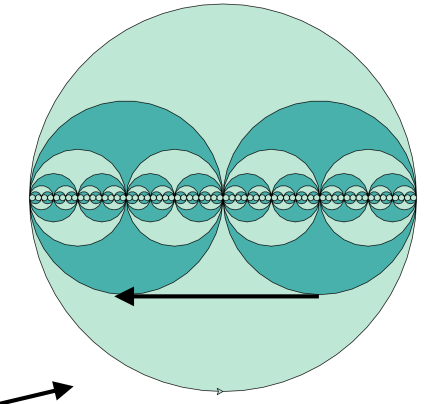
- Wrap up our discussion on graphical recursion
  - Learn about "fruitful" graphical recursion
- Start discussing our next topic: **classes** and **objects**
  - Python is an **object oriented programming** (OOP) language
  - Everything in Python is an **object** and has a **type**
- Learn how to define our own **classes** (**types**) and **methods**

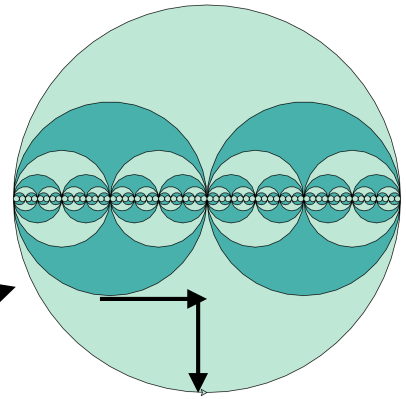
Recap: Nested Circles

- Remember to move turtle back to starting position to maintain **invariance**

```
def nestedCircles(radius, minRadius, colorOut, colorAlt):
    if radius < minRadius:
        pass # do nothing
    else:
        # contribute to the solution
        drawDisc(radius, colorOut)

        # save half of radius
        halfRadius = radius/2

        # position the turtle at the right place
        lt(90); fd(halfRadius); rt(90); fd(halfRadius)

        # draw right subcircle recursively
        nestedCircles(halfRadius, minRadius, colorAlt, colorOut)

        # position turtle for left subcircle
        bk(radius)

        # draw left subcircle recursively
        nestedCircles(halfRadius, minRadius, colorAlt, colorOut)

        # bring turtle back to start position
        fd(halfRadius);  lt(90);  bk(halfRadius);  rt(90)
```



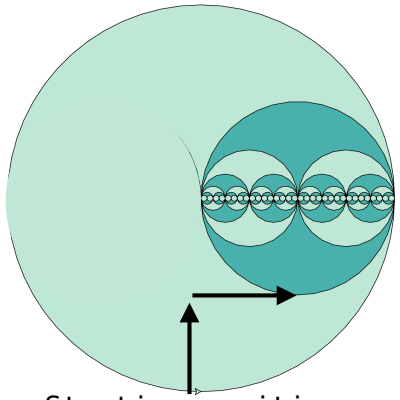


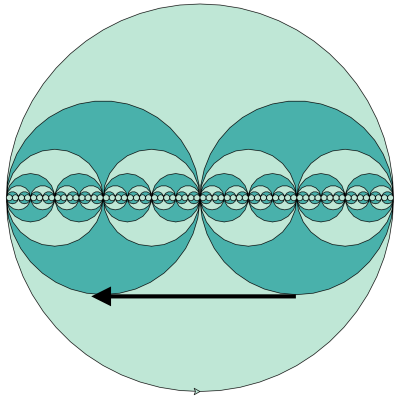


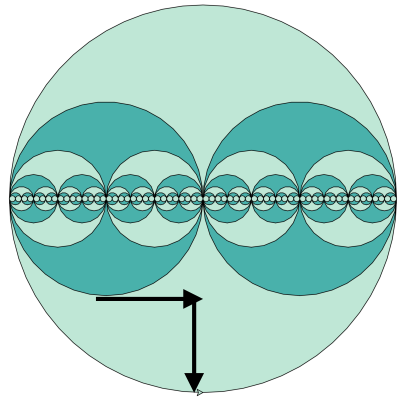
Recap: Nested Circles

- Swapping order of colors in recursive calls facilitates alternating color patterns

```
def nestedCircles(radius, minRadius, colorOut, colorAlt):
    if radius < minRadius:
        pass # do nothing
    else:
        # contribute to the solution
        drawDisc(radius, colorOut)

        # save half of radius
        halfRadius = radius/2

        # position the turtle at the right place
        lt(90); fd(halfRadius); rt(90); fd(halfRadius)

        # draw right subcircle recursively
        nestedCircles(halfRadius, minRadius, colorAlt, colorOut)

        # position turtle for left subcircle
        bk(radius)

        # draw left subcircle recursively
        nestedCircles(halfRadius, minRadius, colorAlt, colorOut)

        # bring turtle back to start position
        fd(halfRadius);  lt(90);  bk(halfRadius);  rt(90)
```

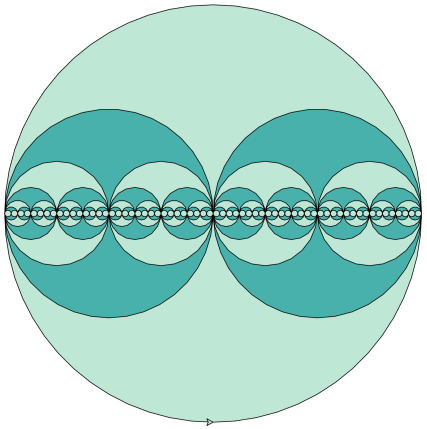






Fruitful Version: Nested Circles

- Suppose we want to keep track of the total number of circles of each color
- Recall that when we explicitly return a value from a function, we call it a "fruitful" function
- Thus in **fruitful recursion**, we need ***explicit*** return statements
- In this case, we should return a **tuple of values**:
  - first item in tuple is # of circles of `colorOut`
  - second item in tuple is # of circles of `colorAlt`



```
nestedCircles(radius, minRadius, colorOut, colorAlt)
```

- `radius`: radius of the outermost circle
- `minRadius`: minimum radius of any circle
- `colorOut`: color of the outermost circle
- `colorAlt`: color that alternates with colorOut

Fruitful Version: Nested Circles

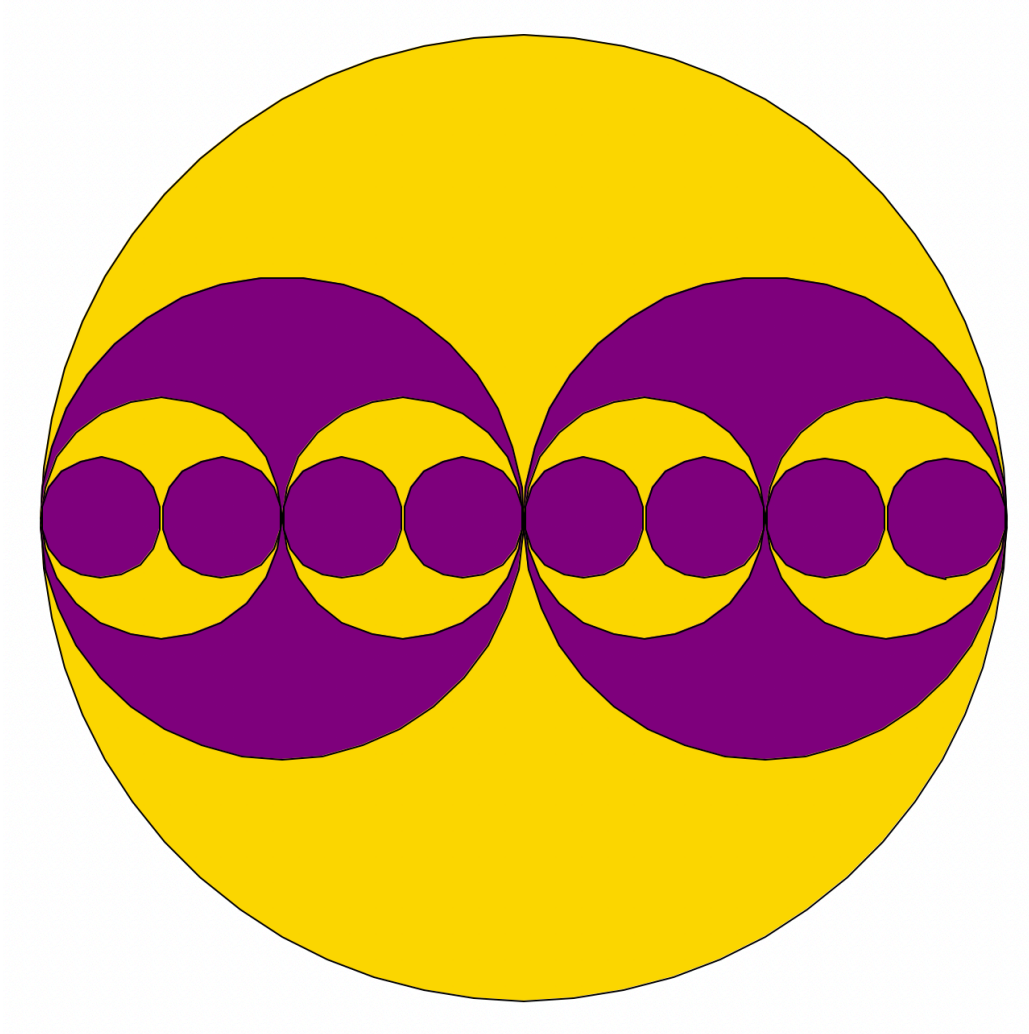
- How does this change our function?
- What should our base case return?
- How do we keep track of number of circles of each color drawn by recursive calls?

```
def nestedCircles(radius, minRadius, colorOut, colorAlt):
    if radius < minRadius:
        pass # do nothing
    else:
        # contribute to the solution
        drawDisc(radius, colorOut)

        # save half of radius
        halfRadius = radius/2

        # position the turtle at the right place
        lt(90); fd(halfRadius); rt(90); fd(halfRadius)

        # draw right subcircle recursively
        nestedCircles(halfRadius, minRadius, colorAlt, colorOut)

        # position turtle for left subcircle
        bk(radius)

        # draw left subcircle recursively
        nestedCircles(halfRadius, minRadius, colorAlt, colorOut)

        # bring turtle back to start position
        fd(halfRadius);  lt(90);  bk(halfRadius);  rt(90)
```

```
def nestedCircles(radius, minRadius, colorOut, colorAlt):
    if radius < minRadius:
        # since we do not draw any circles, we return (0,0)
        return (0,0)
    else:
        # contribute to the solution
        # note that we are drawing one circle with color colorOut
        drawDisc(radius, colorOut)

        # save half of radius
        halfRadius = radius/2

        # position the turtle at the right place
        lt(90); fd(halfRadius); rt(90); fd(halfRadius)

        # draw right subcircle(s) recursively
        # we need to save the resulting tuple
        # order of resulting tuple values corresponds to order of arguments
        rightColorAlt, rightColorOut = nestedCircles(halfRadius, minRadius, colorAlt, colorOut)

        # position turtle for left subcircle
        bk(radius)

        # draw left subcircle recursively
        leftColorAlt, leftColorOut = nestedCircles(halfRadius, minRadius, colorAlt, colorOut)

        # bring turtle back to start position
        fd(halfRadius);  lt(90);  bk(halfRadius);  rt(90)

        # finally, let's return the total sum of the circles
        # values should be ordered colorOut, colorAlt
        return (rightColorOut + leftColorOut + 1, rightColorAlt + leftColorAlt)
```

One call to drawDisc

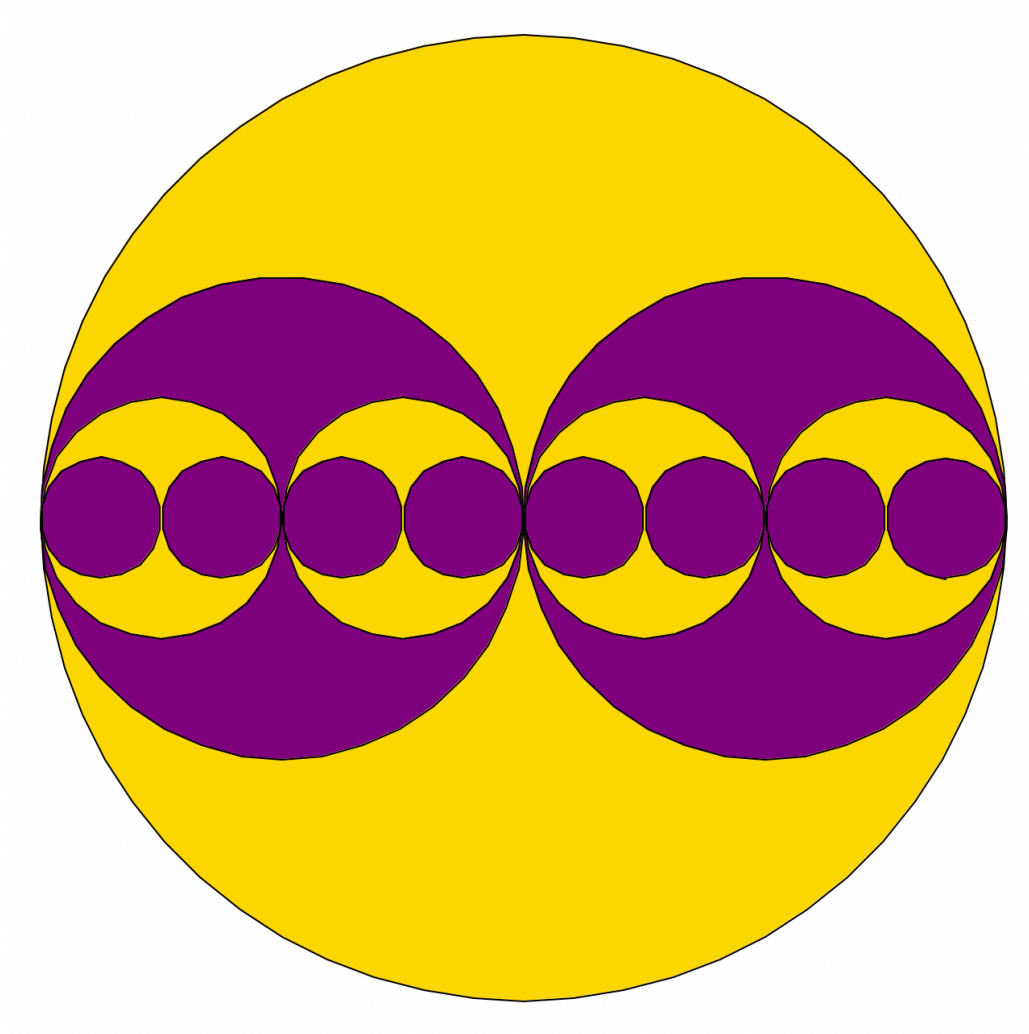
Fruitful Version: Nested Circles



Checking our answers: What do we expect to get?

```
>>> print(nestedCircles(300, 30, "gold", “purple"))
```

Fruitful Version: Nested Circles



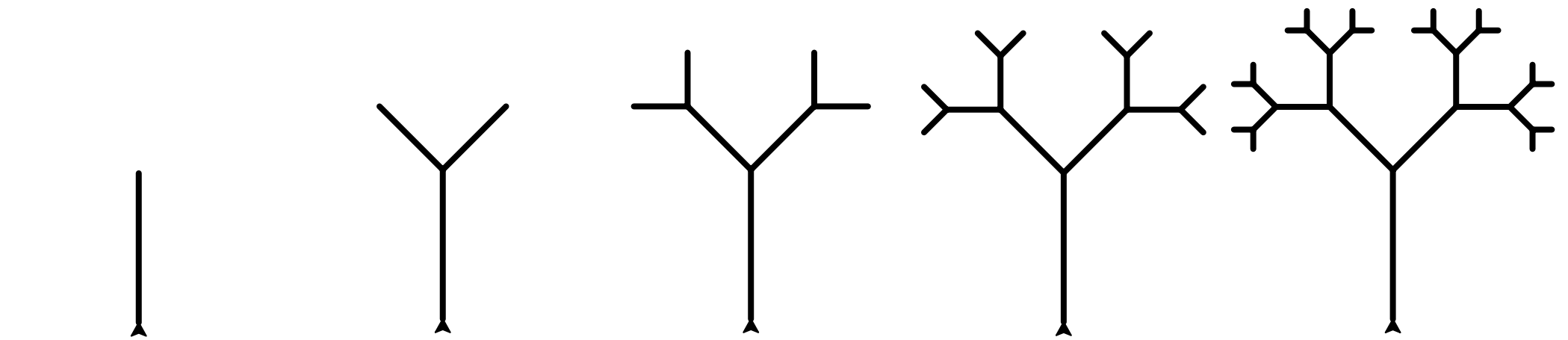
Checking our answers: What do we expect to get?

```
>>> print(nestedCircles(300, 30, "gold", “purple"))
(5, 10)
```

One more Example: Trees

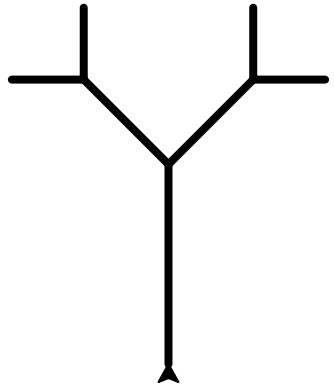
- We can draw more than just circles!
- Suppose we want to draw recursive trees
- What is our base case? Recursive case?
- Note: Assume turtle starts facing north

```
# trunkLen is the trunk length of the main (vertical) trunk
# angle is the branching angle, or the angle between a trunk and its
  right or left branch
# shrinkFactor specifies how much smaller each subsequent branch is in length
# minLength is the minimum branch length in our tree
```



Tree

```
def tree(trunkLen, angle, shrinkFactor, minLength):
    if trunkLen < minLength:
        pass # do nothing

    else:
        # Draw the trunk
        fd(trunkLen)

        # Turn and draw the right subtree.
        rt(angle)
        newTrunkLen = trunkLen*shrinkFactor
        tree(newTrunkLen, angle, shrinkFactor, minLength)

        # Turn and draw the left subtree.
        lt(angle * 2)
        tree(newTrunkLen, angle, shrinkFactor, minLength)

        # Return to starting position
        up(); rt(angle); bk(trunkLen); down()
```



Moving on…Objects in Python

- We have seen many ways to store data in Python

```
1234 3.14159  "Hello" [1, 5, 7, 11, 13] (1, 2, 3)
{"CA": "California", "MA": "Massachusetts"}
```

- Each of these is an **object**, and every object in Python has:
  - a **type** (int, float, string, list, tuples, dictionaries, sets, etc)
  - an internal **data representation** (primitive or composite)
  - a set of functions/methods for **interacting** with the object
- Vocab: A specific object is an **instance** of a type
  - `1234` is an instance of an **int**
  - `"Hello"` is an instance of a **string**

type(object)

- The `type()` function returns the data type for an object

```
In [1]: type(1234)
Out[1]: int

In [2]: type("hello")
Out[2]: str

In [3]: type([1, 5, 7, 11, 13])
Out[3]: list

In [4]: type(range(5))
Out[4]: range
```

```
>>> type(1234)
<class 'int'>
>>> type("hello")
<class 'str'>
>>> type([1, 5, 7, 11, 13])
<class 'list'>
>>> type(range(5))
<class 'range'>
```

- The outputs to
notebook and i

Objects and Types in Python

EVERYTHING IN PYTHON IS AN OBJECT (AND HAS A TYPE)

- Even functions are a type!
- Guido designed the language according to the principle "first-class everything"

```
>>> def greeting():
...     print("Hello")
...
>>> type(greeting)
<class 'function'>
>>>
```

"One of my goals for Python was to make it so that all objects were "first class." By this, I meant that I wanted all objects that could be named in the language (e.g., integers, strings, functions, classes, modules, methods, and so on) to have equal status. That is, they can be assigned to variables, placed in lists, stored in dictionaries, passed as arguments, and so forth." — ***Guido Van Rossum***
(Blog, The History of Python, February 27, 2009)

Stepping Back: Object-Oriented Programming (OOP)

- Python is an **"object-oriented" language**
  - We have been hinting at this aspect all semester
  - Today we will embrace it!
- **OOP** (object oriented programming) is a fundamental programming paradigm
- It has four major principles:
  - **Abstraction** (data and procedural)
  - **Inheritance**
  - **Encapsulation**
  - **Polymorphism**
- We'll explore some of these principles in more detail in the coming lectures

What are Objects?

- It's time to *formally* define **objects** in Python
- Objects are:
    - collections of data (variables or **attributes**) and
    - **methods** (functions) that act on those data
- Example of **abstraction**:
    - Abstraction is the art of hiding messy details
    - Methods define behavior but hide implementation and internal representation of data
        - Eg., You have been using methods for built-in Python data types (lists, strings, etc) all semester without really knowing how the methods are implemented

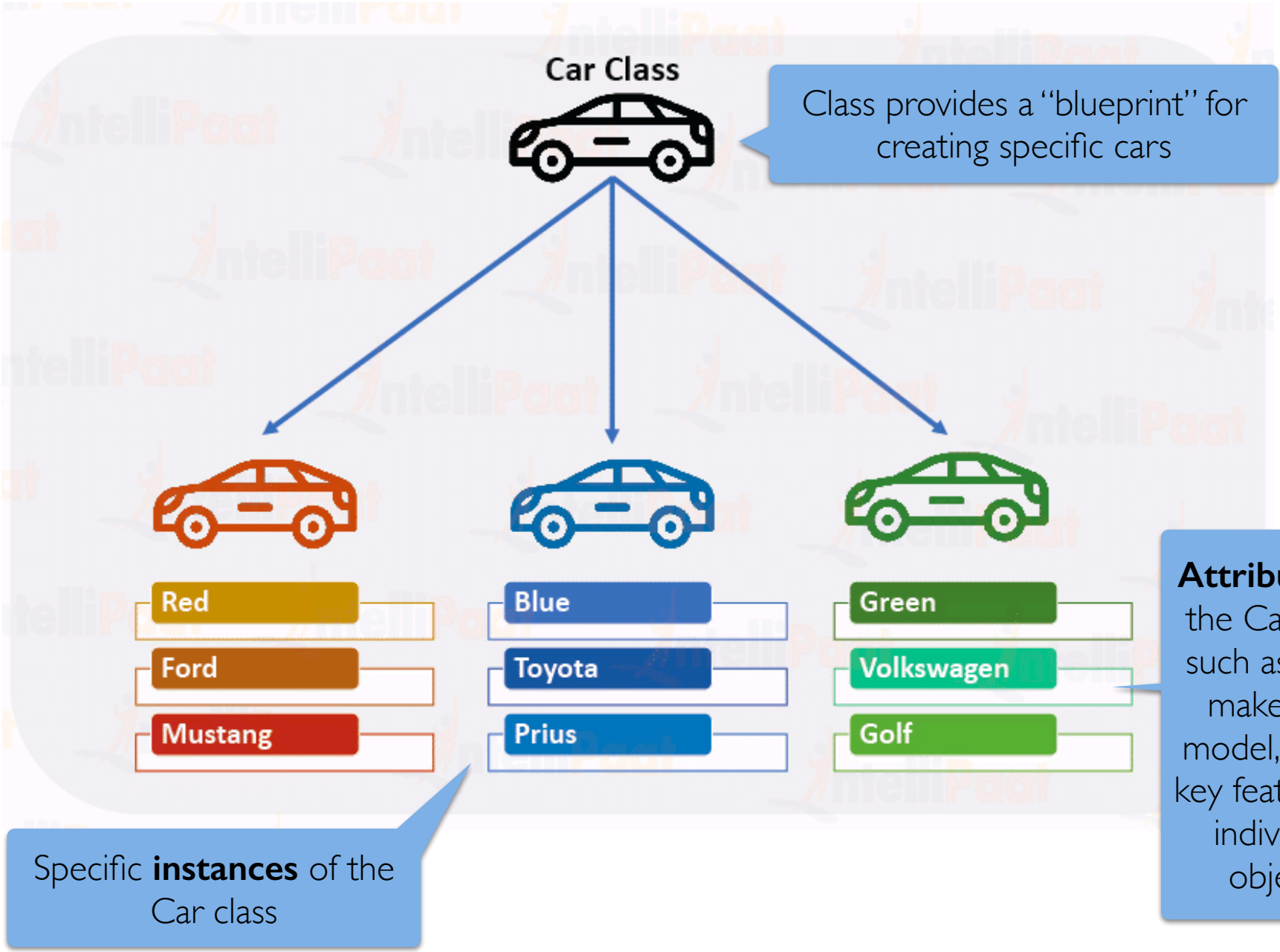
Example: `[1,2,3,4]` has type `list`

- We don't know how Python actually stores lists internally
- Fortunately the typical Python programmer does not need how lists are stored to use list objects (we've been doing it all semester!)
- How do we manipulate lists? Using the methods provided by Python.
  - `myList.append()`, `myList.extend()`, etc.
- **Take away:** Internal representation of objects should be hidden from users. Objects are manipulated through associated **methods**.

Creating Our Own Types: Classes

- It's time to move beyond just the built in Python objects!
- We can create our own data types by defining our own **classes**
  - Classes are like blueprints for objects in Python
- **Creating** a class involves:
  - Defining the **class name**, **attributes, methods**
    - E.g., someone wrote the code to implement a `list` class that we've been using all semester
- **Using** the class involves:
  - Creating **new instances** of the class (which create specific objects)
    - E.g., `myList = [1, 2], myOtherList = list("abc")`
  - Performing operations on the instances through methods
    - E.g., `mylist.append(3)`

Defining Our Own Type: Car class



Defining Methods of a Class

- Methods are defined as part of the class definition and describe how to interact with the class objects
- Example: Recall the following methods for the list class

```
In [1]: L = list()

In [2]: L.extend([1,2,3])

In [3]: L
Out[3]: [1, 2, 3]

In [4]: L.append(4)

In [5]: L
Out[5]: [1, 2, 3, 4]
```

dot operator to "call" the method on the object

Defining Methods of a Class

- On the previous slide, we called methods like `append()` and `extend()` on a particular list **object** L.
- We can define methods in our classes in a similar way
- Consider this simple example:

```
In [6]: class A:
            """Class to test the use of methods"""
            def greeting(self):
                print("Hello")
```

Defining Methods of a Class

- To create methods that can be called on an instance of a class, they must have a parameter which takes the instance of the class as an argument
- In Python, by convention, the **first parameter is used as a reference to the calling instance**. This parameter is usually called `self`.

```
In [6]: class A:
            """Class to test the use of methods"""
            def greeting(self):
                print("Hello")
```

All methods include the `self` parameter.

Our First Method

```
In [1]: class A:
            """Class to test the use of methods"""
            def greeting(self):
                print("Hello")
```

- How do we call the greeting method?
  - We create an instance of the class and call the method on that instance using dot notation:

```
In [2]: a = A()
```

```
In [3]: a.greeting()
```

```
Hello
```

Mysterious `self` Parameter

- Even though method definitions have `self` as the first parameter, **we don't pass this parameter explicitly** when we invoke the methods
- This is because whenever we call a method on an object, the object itself is **implicitly** passed as the first parameter
- Note: In other languages (like Java) this parameter is implicit in method definitions but in Python it is explicit and by convention named **`self`**
- **Take away:**
  - When defining methods, always include self
  - When calling or invoking methods, the value for self is passed implicitly

Summary of Classes and Methods

- **Classes** allow us to define our own data types
- We create **instances** of classes and interact with those instances using methods
- All **methods** belong to a class, and are defined within a class
- A method's purpose is to provide a way to access/manipulate **objects** (or specific instances of the class)
- The first parameter in the method definition is **the reference to the calling instance (self).**
- When **invoking** methods, this reference is provided implicitly