Advanced Programming #2: Language Types

CS SCHOLARS - PROGRAMMING

Learning Goals

Understand the different types of **imperative** and **declarative** programming paradigms

Write Python code in an **object-oriented** style and in a **functional** style

Recognize other **variations** in programming language design have strengths and weaknesses

Programming Paradigms

There are many different ways to approach programming. To start, we'll examine two main categories of programming languages.

Imperative programming languages are **state-based**. In programming, we think of state as the data that a program processes, the current values held in the computer's memory that can be accessed or changed. State-based programming revolves around the state and how to change it.

Declarative programming languages are **property-based**. These languages avoid changing the state of the program; instead, they declare what the result of a computation should look like. These properties are used to perform actual computation.

Imperative Programming

Imperative Programming

When programming imperatively, you usually focus on **how** the program should change the state over time. This is done with direct commands on variables and other data structures that might update over time.

There are many different sub-categories of imperative programming languages. Two common ones are **procedural** and **object-oriented** programming, which are two different ways to organize the commands of programming.

Imperative – Procedural

Procedural programming revolves around describing different procedures (or **functions**) that can be performed on data, then calling those procedures on data directly.

Most of the programming we'll do in Python is procedural, though Python can be used in other programming paradigms as well. Another common procedural language is C.

Imperative – Object-Oriented

Object-oriented programming (or OOP) still uses functions, but groups the functionality of the program into different **objects** to improve organization.

A classic theme in object-oriented programming is **inheritance**- build objects to inherit features and methods from each other, so that repeated work can be minimized.

One of the most common object-oriented languages is **Java**. Python is not quite as object-oriented as Java, but you can still do OOP in Python if you want to!

Programming in Python with OOP

You can try creating and using objects yourself with just a little more Python syntax!

The `class` command lets you set up a new **class definition**. This describes a new object type that you design, both the properties of that type and the actions it can take.

```
class MyObject:
    # stuff goes here
```

Class definitions use indentation to specify what is part of the definition vs. not the same way that function definitions do.

OOP Properties in Python

First, you can add **properties** to a class to describe attributes that are represented in it.

To add a property to a class, put a variable assignment into the indented code under the class line. When you want to stop adding properties, un-indent the code to go back to normal code.

```
class Coordinate:
    x = 0
    y = 0
```

Programming in Python with OOP

Once you've defined a class, you can create a new **instance** of that class by calling the class name as if it were a function.

```
point = Coordinate()
```

If the class defines the general properties of this data type, the instance, or **object**, represents a specific example of that type.

You can also use the properties of the object as if they were variables. To do this, use the object variable, then a period, then the name of the property you want to access.

```
print(point.x) # originally 0
point.x = 20
print(point.x) # now it's 20!
```

Programming in Python with OOP

The cool thing about objects is that you can create as many objects as you want, and they're all **separate**. Changing one doesn't change the others.

```
point1 = Coordinate()
point2 = Coordinate()
point1.y = 10
print(point2.y) # still 0!
```

Methods vs Functions

You can also add **methods** to a class to describe what that class can do, and what can be done to it.

A method is like a function, except it must be called **on** an object instead of being called directly in the code. You can call some methods on strings, for example.

We'll learn more about methods when we learn about lists.

```
abs(10) # example of a function
"hello".isalpha() # example of a method
```

OOP Methods in Python

To set up a method in a class definition, just define a function indented inside the class.

You do have to add one special component, an extra parameter at the front of the parameter list, typically called `self`. This parameter will refer to the object that the method is called on.

You can use the same period syntax to access the properties of the `self` object.

```
class Coordinate:
    x = 0
    y = 0

    def distance(self, otherX, otherY):
        xDiff = (self.x - otherX)**2
        yDiff = (self.y - otherY)**2
        return (xDiff + yDiff)**0.5
```

Set Up Objects with `__init__`

Usually, we don't set up the properties of an object directly in the class. Instead, we set them up in a special method called `__init__`.

This method is called when the instance is constructed. It lets you set up specific property values for each new object.

There are a lot more specialty methods you can implement in a class. Learn more at https://docs.python.org/3/reference/datamodel.html

```
class Coordinate:

    def __init__(self, x, y):

        self.x = x

        self.y = y


point1 = Coordinate(10, 10)

point2 = Coordinate(3, 5)

print(point1.x, point2.x) # 10 3
```

OOPy Programming Example

Here's an example of how we might calculate the distance between two points in an OOPy way.

OOP is mainly useful when you're working with lots of intricate data types. It's great for massive application development, and game development.

```
class Coordinate:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self, other):
        xDiff = (self.x - other.x)**2
        yDiff = (self.y - other.y)**2
        return (xDiff + yDiff)**0.5

point1 = Coordinate(10, 10)
point2 = Coordinate(3, 5)
print("Distance:", point1.distance(point2))
```

Declarative Programming

Declarative Programming

When programming declaratively, you focus on **what** the result should look like, in terms of its properties. When the computation is expressed directly, the programming language itself can evaluate the code to determine what the result should be.

Many declarative languages aim to **avoid changing state**, and instead create new state when needed. This is done to avoid side effects, and to make it possible to mathematically prove when code works.

There are many different subfields of declarative programming. We'll look into two- **functional** and **logic** programming.

Declarative – Functional

Functional programming uses functions, like procedural programming. However, these functions do not track state; they derive the output directly based on the input.

This is often done by stating the returned value recursively, where the function calls itself on a different input. This is similar to a proof by induction- if we can derive f(x-1), we can derive f(x). Recursion will show up in a future advanced topic.

A common functional programming language is Haskell. You can also write code functionally in Python as long as you avoid changing the program state.

Functional Programming in Python

The concepts we've learned so far are actually pretty easy to fit into functional programming. We just need to avoid changing variables to take a functional approach.

One way to ensure that you don't change state is to define functions as **lambda functions** instead of regular function definitions. A lambda function is written all on one line, and the 'body' of the function evaluates down to a single expression (the value to be returned). It takes the format:

`lambda` *args* `:` *returnedValue*

For example, to calculate the distance between two points functionally, we could write the following lambda.

```
distance = lambda x1, y1, x2, y2 : ((x2 - x1)**2 + (y2 - y1)**2)**0.5
distance(1, 1, 4, 5) # 5.0
```

Functional Programming with Lists

Functional programming mainly looks different from regular programming when it comes to **iteration** and **lists**, which we'll cover in the next two weeks.

If you want to peek ahead, try reading up on some of the core functions of functional programming:

`map` – takes a list and applies an operation to every value in that list
- https://docs.python.org/3/library/functions.html#map

`reduce` – takes a list and combines all the elements of the list together based on an operation
- https://docs.python.org/3/library/functools.html#functools.reduce

`filter` – takes a list and filters out values that don't match a requirement
- https://docs.python.org/3/library/functions.html#filter

Declarative - Logic

Logic programming sets up a series of logical facts and rules and uses those facts and rules to derive new ideas. When the user sets the system a goal, the system attempts to achieve the goal by chaining together the facts and rules already known.

This is mainly used in mathematical settings, to derive new proofs. However, math can be applied in many fields of computer science, especially machine learning. This is also useful when attempting to find a solution that meets a certain set of constraints.

Python does not directly support logic programming, but there are external packages which can be imported in Python to perform these kinds of operations. A common logic programming language is Prolog.

Language Variations

Other Variations

Beyond Imperative and Declarative language styles, there are dozens of other models that programming languages can use to support different programming tasks. A list can be found here.

Beyond that, programming languages make many choices about how to represent syntax and process code. All of this variation means that there are hundreds of programming languages to choose from!

We'll look at a few different options that you might have noticed when comparing Python to a language you've learned before: **compiled** vs. **dynamic**, **weakly typed** vs. **strongly typed**, and **text** vs. **block**.

Compiled vs. Dynamic

When a computer runs a program, it needs to **parse** and **compile** the program before it can compute the result. This is the process the computer uses to understand how code should be executed at the machine level, where commands eventually turn into hardware-level operations.

Some languages are **compiled-** the code must be fully compiled before it can be run. Compilation will often pre-perform some operations, to optimize how quickly the program runs when the user begins the process. Java is a common compiled language.

Other languages are **dynamic-** they re-compile the code every time the user runs it, and can add certain computations in as the program runs. These programs are often slower, as the computer can not pre-calculate results, but they also allow for more experimentation. Python is a common dynamic language.

Weakly vs. Strongly Typed

Most programming languages have a concept of **data types**. We've already gone over ints, floats, strings, and Booleans in this class.

Some languages have **weak typing**. Every variable has a type at runtime, but the type of the variable can change as the variable itself is updated with new values. This allows for more flexibility during code-writing. Python is a common weakly-typed language.

Some languages have **strong typing**. Variables must be assigned a type when they are defined, and the type may not change during runtime. Type changes are considered runtime errors in these languages. This prevents bugs caused by accidental type changes during computation. Java is a common strongly-typed programming language.

Text vs. Block

In all programming languages, the user must communicate the program to the computer in some way. But the modality used for communication can be different across different languages!

Most languages use **text**, like Python. The user must type out the text of the program, then the program parses the text into tokens and evaluates it. The syntax of the text varies across languages.

Some languages use **blocks** instead of text. These languages specify all the different commands of the language as visual blocks, and the user can drag-and-drop them into an environment, arranging and nesting them as needed to achieve functionality. These blocks directly represent tokens, so syntax errors are rarer. Scratch is a common block language.

Learning Goals

Understand the different types of **imperative** and **declarative** programming paradigms

Write Python code in an **object-oriented** style and in a **functional** style

Recognize other **variations** in programming language design have strengths and weaknesses

Most of the information in this slide deck was developed based on Wikipedia's entry on Programming Paradigms. If you're interested, you can read more there!