The Python Programming Language

History of Python

From `grep` to `sed` to `awk`, such commands attempt to generalize and expand what the previous does, rising in levels of complexity.

The **Perl** programming language was designed to be able do everything awk, etc. can do.

Then **Python** was designed to do everything Perl, etc. can do. The rise of Python can be attributed to two parts:

Part 1: BASIC and ABC

BASIC: an instructional language. Instructors noticed that students showed up knowing BASIC.

Instead of writing very low-level code, go up one level of abstraction. Give people a language where:

- Hash tables are implemented into the language, like `set` and `dict`
- All the basic algorithms already built-in. Just `sort` lol.
- Enforce indentation.
- An IDE to run the programs.

High school students will then be programming in this new language called **ABC**. It didn't work because employers still wanted BASIC.

Part 2: The Perl Programming Language

> Perl = sh + awk + sed + ...

Perl was designed as an antidote to the "Little Languages" philosophy, so it combined all the little languages into one scripting language. Perl became the scripting language of choice for about 10 years. It was designed to be like a real spoken language - there was always more than one way to do something.

A Combination

However, ABC had the philosophy that there is one correct way to do anything.

Python emerged as a combination of:

- The philosophy of ABC. From The Zen of Python:

  > There should be one-- and preferably only one --obvious way to do it.

- The capabilities of Perl. Python is a scripting language that tries to do everything. Theoretically, if you know the language very well, you do not have to touch the little languages of the shell. Python crutches FTW 😄 (very not biased)

> **ASIDE:** All 3 languages, BASIC, Perl, and Python can be either compiled or interpreted.

Why Python?

- **CONS:** Slow, memory hog.
- **PROS:** Easier to write (development cost vs. runtime cost). A lot of libraries are also written in C/C++ code for a performance bonus, made possible by **native method interfaces**.

Scripting languages like Python demonstrate an alternate balance between **development cost** and **runtime cost**. Often times, human time is much more valuable than computer time.

Prevalent in machine learning, although this feat is probably due to being at the right place at the right time. Python happened to be a reasonable scripting language for the job when the field was emerging in popularity.

Oh yeah and here's a quote I found pretty *profound*:

> There's always going to be a time where you're the blind person next to the elephant. The goal of software construction is to make you a better blind person. **- Dr. Eggert**

Python Internals

Python is **object-oriented**. Historically, it didn't actually start out that way. It started with functions but no classes. When classes were introduced, they implemented *methods* as functions that explicitly take the `self` first argument, which is actually how OOP is implemented behind the hood in languages like C++.

Anyway, every value is an object. Every object has:

- Identity - *cannot be changed*
- Type - *cannot be changed*
- Value - *can* be changed, but only if the object is **mutable**

Typing

In old Python, `int` used to have fixed size, so there was the distinction between integers and longs.

The main *categories* of types in modern Python:

Singletons: Types with only one instance throughout runtime

- `None`: Python's version of a `null` value
- `True` and `False`: Python's booleans, which (fun fact) actually subclass the numeric type `int`.

Numbers

- `int`: A number without a fractional part. In modern Python, these can be any size, so you don't need to worry about bounds/overflow like in most languages.
- `float`: A number with a fractional part. There's also the special `float("inf")` constant that represents infinity (`1/0` in C, `Infinity` in JavaScript, etc.).
- `complex`: A number with a real and imaginary part. You can initialize a literal with the `a+bj` syntax e.g. `5+4j`.

Collections

- **Sequential Containers**: Collections where order matters, with elements being accessed by **index**
  - `list`: Python's builtin vector type. Heterogenous, arbitrary-sized, ordered collections.
  - `tuple`: Same as `list`, but immutable, so it's fixed length. These are nice because they are more efficient and are in a sense "safer" to use.
- **Associative Containers**: Collections where elements are accessed by **key**
  - `set`: Python's builtin hash table. Unordered collection of unique, **hashable** values.
  - `dict`: Python's builtin hash map. Unordered collections of key-value pairs. Keys must be unique and **hashable**.

Callables: Objects that support being called with optional arguments

- **Functions**: objects you define with the `def` keyword or anonymous ones with the `lambda` keyword.
- **Methods**: function objects that have been bound to an instance of a class. They take a mandatory `self` positional argument, the class instance the method acts on behalf of.

There's also the **buffer** type. I assume this is only available in Python 2. Buffers are like multiple strings. When you're done working with it, you can convert it to a string with `str(x)`.

Lists

Common `list` methods:

- `my_list.append(value)`: add any value as an element to the end of the list.
- `my_list.extend(iterable)`: lay all the values out from the iterable as elements at the end of the list.
- `my_list.insert(index, value)`: insert an element at a position in the list, pushing everything after it backwards.
- `my_list.count(value)`: return the number of occurrences of `value` (using `==` checking).
- `my_list.remove(value)`: remove the first occurrence of `value` (using `==` checking), raising `ValueError` if not found.
- `my_list.pop([index])`: defaults to last item, which you can use along with `.append` to emulate a stack data structure. Raises `IndexError` if empty.
- `my_list.clear():` remove all elements.

Common operations universal to sequential containers:

- `len(my_list)`: return the number of elements.
- `my_list[i:j]`: return a `slice` of the container, starting from index `i` and up until but excluding `j`.
  - Mutable ones also support this syntax on the LHS, where it means **reassigning** a segment of the container, as well as `del my_list[i:j]`, which deletes that segment of the container.

ASIDE: Underying `list` Allocation

- Probably uses cache size to determine starting size.
- After that, reallocation uses geometric resizing (approximately nine-eights according to mCoding).
- The total cost of calling `list.append` N times is $O(N)$. Because the **amortized cost** of this operation is $O(1)$.

Visualization: imagine that the list length is doubled for every allocation, which isn't true, but this doesn't change the asymptotic time, so it simplifies the derivation:

```
[e e e e e e e e e e e e e e e e e e e e e e]
  ... |<3>|<--2 ops-->|<-------1 op-------->|
```

The total cost is $\frac{1}{2} * 2 + \frac{1}{4} * 2 + \frac{1}{8} * 3 + \ldots$, which converges to 2, which is $O(1)$.

Python vs. Shell vs. Emacs Scripting

Lisp is an ASL for Emacs, like an extension language. It uses existing code, *Emacs primitives*.

Shell uses existing programs.

Python was designed to be a *general-purpose programming language*, so there are no "primitives" you bring together - you just write in the language altogether to program from scratch. However, it also converges to the same phenomenon where programmers glue together existing modules like PyTorch and SciPy.

What makes a language a scripting language is one that supports this pattern of software construction of building applications from existing code.

Another quote I found quite *profound*:

> The goal of a **scripting language** is you don't code from scratch. You glue together other people's code. You provide the cement, and the other people provide the bricks. **- Dr. Eggert**

Classes and OOP

Class hierarchies are **directed acyclic graphs (DAG)**. This is especially apparent because unlike languages like Java, Python supports **multiple inheritance**.

Python's **method resolution order (MRO)** is depth-first, left-to-right. So for example, if you define a class that inherits like so:

```python
class C(A, B):
    def some_method(self, arg):
        pass
```

With the DAG model, this design makes it so that if `A` and `B` disagree, `A` will always take priority.

Historical ASIDE: The decision to explicitly include `self` in all method definitions was to not abstract a fundamental mechanism of OOP: every method is *bound* to the class and *acts on* the instance. If you examine the machine code of similar OOP languages like C++, you'll see that there's a hidden first argument to every method, that is the pointer to the object that the method is acting *on behalf of*.

Introspection ASIDE: You can use the built-in `__mro__` attribute of class objects to programmatically access a class' method resolution order. For example:

```python
# Print the names of the classes in class O's MRO
print([cls.__name__ for cls in O.__mro__])
```

Dunders and "Operator Overloading"

Besides the ones you already know...

The old way to redefine the comparison operators:

```python
def __cmp__(self, other):
    # negative for <, 0 for equal, positive for >
    return num
```

This is still supported but it is now anachronistic approach because you can run into hardware problems. A notable example is the case of *floating point numbers*, which have an additional state, **NaN**, beyond negative, zero, and positive. Thus, we have the familiar `__lt__`, `__gt__`, etc.

This is the Python 2 predecessor to the familiar `__bool__` method:

```python
def __nonzero__(self):
    # Return whether the object is considered to be "not zero"
    return b
```

Namespaces in Classes

Namespaces are just dictionaries. Classes have a special attribute `__dict__`, a `dict` that maps names to values. This gives rise to opportunities to write "clever" Python code, where you can programmatically alter the definition of an existing class:

```python
c = C()
c.__dict__["m"] is c.m
```

This is (probably) how **metaclasses** are implemented, which was not mentioned in lecture but is cool to know. They're basically classes that determine how other classes should be implemented.

Modules

The `import` Statement

1. Creates a namespace for the module.
2. Executes the contents of the module *in the context of that namespace*. Eggert didn't mention this, but this step is actually only performed if the module *hasn't already been imported*. Modules are only run onces.
3. Adds a name, the module name, to the current namespace.

Proof for #2:

```
# module.py
print("Hello world")
```

```
# runner.py
import module
import module
```

```
$ python3 runner.py
Hello world
```

Packages

Packages organize source code into a familiar tree structure. This allows importing to be parallel the file system.

The special `__init__.py` scripts turns a directory into a proper package, and it is automatically run upon import.

The `PYTHONPATH` Environment Variable

Just as how `PATH` instructs the shell program where to search for commands, `PYTHONPATH` instructs the Python interpreter on where to search for code.

Determines the behavior of the `import` statement. Python will search through the sequence of paths, delimited by colons (Unix) or semicolons (Windows), to search for names of packages or modules to import. The path to the standard library is included in `PYTHONPATH` by default.

Official documentation: https://docs.python.org/3/using/cmdline.html#envvar-PYTHONPATH.

This variable is stored in and can be modified programmatically with `sys.path`, which is a `list[str]` containing the individual string paths.

Why all this complexity with packages vs classes?

Packages are oriented towards developers (like a *compile-time notion*). The tree is structured so that different developers can work on different parts of the code.

Classes are about runtime behavior (a *runtime notion*). You want inheritance to be independent of package hierarchy. Classes are only concerned with their own behavior, "what to do next", so it should be able to pull code from anywhere in the codebase. How developers *organize* that codebase is made possible with packages.