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:

Step 0: FORTRAN

Step 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.

Step 1.5: Visual Basic

Step 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
>
> The analogy to—being a "blind person next to the elephant" is a variation on the well-known parable of the blind men and the elephant. In this story, several blind men touch different parts of an elephant to learn what it is like. Each one feels a different part, such as the side, the tusk, or the trunk, and then they compare their views. Because they each felt only one part, their descriptions of the elephant are completely different from one another. This story is often used to illustrate the point that people can have widely differing perceptions of the same object or concept, depending on their limited experience and perspective.
>
> In the context of software development, this analogy highlights how developers might understand only a part of the overall system or problem they're working on, especially in complex or large-scale projects. Each developer might be familiar with just a segment of the application or a slice of the technology stack, akin to the blind men understanding only the part of the elephant they could touch.
>
> The statement "The goal of software construction is to make you a better blind person" suggests that while it may be unrealistic to expect any one person to comprehend the entirety of a complex system (to "see" the whole elephant), the aim should be to improve developers' ability to understand, interact with, and make decisions about their piece of the system as effectively as possible.

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)`.

Strings

`str.strip()`

1. Default Behavior (No Arguments): When strip() is used without any arguments, it removes all leading and trailing whitespace characters (like spaces, tabs, newlines) from the string.

2. Specifying Characters: You can also specify a string of characters to be stripped. strip() will then remove all combinations of those characters from both the start and the end of the string.

```
s = '   spacious    '
# strip(No_Argument) removes all leading and trailing whitespace
characters (like spaces, tabs, newlines)
print(s.strip()) # Output: "spacious"
```

```
s = 'aaahh   panic    aahhh'
# strip(Argument) remove all combinations of those characters from
both the start and the end of the string
print(s.strip('ah')) # Output: "   panic    "
# remove all combinations of 'a' & 'h'
print(s.strip('ah ')) # Output: "panic"
# remove all combinations of 'a', 'h', ' '
```

`str.rstrip()`

1. This method is used to remove trailing characters (characters at the end of a string).

2. By default, it removes any whitespace characters, such as space, newline (\n), or tab (\t). However, you can specify a string of characters to be removed.

```
text = "  This is a sentence with whitespace at the end.     \n"
cleaned_text = text.strip()
print(repr(cleaned_text))

cleaned_text = text.rstrip()
# "This is a sentence with whitespace at the end."
print(repr(cleaned_text))
# "  This is a sentence with whitespace at the end."
```

```
text = "....This is a sentence with a period at the end...."
cleaned_text = text.strip('.')
print(repr(cleaned_text))
# "This is a sentence with a period at the end"

cleaned_text = text.rstrip('.')
print(repr(cleaned_text))
# "....This is a sentence with a period at the end"
```

`str.join()`

```

```

```
L = ["ba", "na", "na"]
print("".join(L)) # banana
print(" ".join(L)) # ba na na
print("---".join(L)) # ba---na---na
```

`str.split()`

1. Default Use (No Arguments): When used without arguments, split() will treat consecutive whitespace as a single separator, splitting the string at each occurrence of whitespace.
2. Specifying a Separator: You can specify a string as the separator. The method will split the string at each occurrence of this separator.
3. Using maxsplit: If the maxsplit parameter is provided, the method will split the string at most maxsplit times, with the remainder of the string being returned as the last element of the list.

```
text = "hello world welcome to Python"
print(text.split())
# ['hello', 'world', 'welcome', 'to', 'Python']
# use whitespace as separator, splitting the string at each occurrence of
whitespace

text = "apple,banana,cherry"
print(text.split(','))
# ['apple', 'banana', 'cherry']
# use ',' as separator, splitting the string at each occurrence of comma

text = "one:two:three:four"
print(text.split(':', 2))
# ['one', 'two', 'three:four']
# use ':' as separator, splitting the string at most 2 times
```

`str.splitlines()` This method splits a string into a list where each element is a line from the string. It splits the string at line breaks (\r, \n, \r\n). This is useful when you have a string that contains multiple lines and you want to work with each line separately.

```
text = "First line.\nSecond line.\rThird line.\r\nFourth line."
lines = text.splitlines()
print(lines)
# ['First line.', 'Second line.', 'Third line.', 'Fourth line.']
```

`str.replace()`

1. Basic Usage: The method returns a new string where all occurrences of the old substring are replaced with the new substring.

2. Lmiting the Number of Replacements: By using the count parameter, you can limit the number of replacements.

```
text = "I like apples. Apples are great!"
print(text.replace("apples", "oranges"))
# Output: "I like oranges. Oranges are great!"

text = "I like apples. Apples are great! Apples are healthy!"
print(text.replace("apples", "oranges", 1))
# Output: "I like oranges. Apples are great! Apples are healthy!"
```

Dictionary

```
d = {1: "a", 2: "b", 3: "c"}
print(d[1])

print(d.get(1)) # error-free (will not pop error if this key not exist)
print(d.get(4))  # Outputs: # default output is 'None'
print(d.get(4, 'Not Found'))  # Outputs: 'Not Found'

d.setdefault(4, 'default')  # Adds key 4 with value 'default' if key 4 is
not in the dictionary
print(d)

d_copy = d.copy()
d.update() # changes nothing
d.update({2: "c", 5: "d"})

key, value = d.popitem() # Popping an item and print

print(d.clear()) # None
```

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.

Anonymous functions & Lambda

You can define a function without a name if the function definition is a one-liner. This character will be executed by lambda.

```
# function_name = lambda return_value : function_definition
double = lambda x : 2 * x
# def double(x):
#     return 2*x
print(double(3)) # Output: 6

multiply = lambda x, y: x * y
print(multiply(2, 3)) # Output: 6
```

Anonymous function is useful when passed as a function argument. L = [3, 1, 23, 4, 6, 100] L.sort(key=lambda x: x % 2) # [1, 1, 1, 0, 0, 0] print(L)

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 $\mathrm{O(N)}$. Because the **amortized cost** of this operation is $\mathrm{O(1)}$.

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]
  ... |<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 $\mathrm{O(1)}$.

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:

```
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:

```
# 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:

```
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:

```
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:

```
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.

Argparse

The **argparse module** makes it easy to write user-friendly command-line interfaces and allows the programmer to define what arguments are required, and Python will figure out how to parse those out of sys.argv.
The argparse module also automatically generates help and usage messages and issues errors when users give the program invalid arguments.

- How argparse works
  When you create a parser, you tell it what arguments to expect. Each argument can trigger different actions, and you can specify the type of data you expect (like integers, strings, etc.). When the script is run, argparse processes the arguments user provided on the command line, converts them to the specified type, and then your script can use them.

```
import argparse

# Create the parser
parser = argparse.ArgumentParser(description='Example script')

# Add arguments
parser.add_argument('name', help='Your name')
parser.add_argument('--age', type=int, help='Your age')

# Parse the arguments
args = parser.parse_args()
'''
# parse_args(): This method of the ArgumentParser instance parser is
responsible for interpreting the arguments passed to the script when it's
run from the command line. It reads the arguments from sys.argv (the list
of command-line arguments passed to a Python script), parses them, and
returns an object (typically named args) containing the arguments and
their values.
'''

# Use the arguments
print(f"Hello {args.name}, you are {args.age} years old.")
```

> Use Emacs to write a script shuf.py. Your script should implement the GNU `shuf` command that is part of GNU Coreutils. GNU `shuf` is written in C, whereas you want a Python implementation so that you can more easily add new features to it.
>
> Your program should support the following `shuf` options, with the same behavior as GNU shuf: **--echo (-e), --input-range (-i), --head-count (-n), --repeat (-r), and --help**. As with GNU shuf, if --repeat (-r) is used without --head-count (-n), your program should run forever.
>
> Your program should also support *zero non-option arguments* or *a single non-option argument "-" (either of which means read from standard input)*, or *a single non-option argument other than "-" (which specifies the input file name)*. As with GNU `shuf`, your program should report an error if given invalid arguments.

```
#!/usr/local/cs/bin/python3
import argparse
import random
import sys

def generate_input(args):
    if args.echo is not None:
        return args.echo
    elif args.input_range:
        start, end = map(int, args.input_range.split('-'))
        return [str(i) for i in range(start, end + 1)]
    elif args.file == '-':
```

```
            return [line.strip() for line in sys.stdin]
        else:
            with open(args.file, 'r') as f:
                return [line.rstrip('\n') for line in f]

    def main():
        parser = argparse.ArgumentParser(description="Generate random
    permutations of input lines")
        parser.add_argument('-e', '--echo', nargs='*', help="Treat each ARG as
    an input line")
        parser.add_argument('-i', '--input-range', help="Treat each number LO
    through HI as an input line")
        parser.add_argument('-n', '--head-count', type=int, help="Output at
    most COUNT lines")
        parser.add_argument('-r', '--repeat', action='store_true',
    help="Repeat output values")
        parser.add_argument('file', nargs='?', default='-', help="Input file
    or '-' for standard input")

        args = parser.parse_args()

        # Validate head count
        if args.head_count is not None and args.head_count < 0:
            parser.error("invalid head count: cannot be negative")

        # Validate input range format
        if args.input_range:
            try:
                start, end = map(int, args.input_range.split('-'))
                if start > end:
                    parser.error("invalid input range: start should not be
    greater than end")
            except ValueError:
                parser.error("invalid input range format: expected format is
    'start-end'")

        lines = generate_input(args)

        if args.head_count is not None:
            lines = lines[:(int(args.head_count))]

        random.shuffle(lines)

        if args.repeat and not args.head_count:
            while True:
                print(random.choice(lines))
        else:
            for line in lines:
                print(line)

    if __name__ == "__main__":
        main()
```

> `update()`
>
> In Python, strings are immutable, which means they cannot be changed after they are created. Therefore, there is no update() method for strings. Any operation that modifies a string will actually create a new string, rather than changing the original one.
>
> As for lists, there is no update() method for lists either. The update() method is actually a method for dictionaries, not lists. For lists, you typically use methods like append(), extend(), or insert() to add items, and remove(), pop(), or slicing to remove items.

```
my_dict = {'a': 1, 'b': 2}
my_dict.update({'b': 3, 'c': 4})  # Updates 'b' and adds 'c'
# my_dict would be {'a': 1, 'b': 3, 'c': 4}.
```