Python

```
import tensorflow as tf
```

Overview

- High level scripting
- What is Python?
- Fundamentals
  - Variables
  - Types
  - Expressions
  - Statements
- Modules and packages and the standard library
  - Package managers
- Useful tidbits
- Extra
  - Debugging, NumPy, SciPy, Matplotlib

High level scripting

- Shell scripting syntax is rather unwieldy
  - It's oriented around organizing running utilities
- Traditional compiled high-level languages (C, C++, Java, etc.) tend to have a lot of boilerplate to deal with
  - They go fast though
- What if you want something easy and powerful but don't necessarily need blazing performance?
  - This is where higher level programming/scripting languages come in
  - Python, Perl, Ruby, to name a few
  - Tend to be interpreted, not needing compilation
  - Often come with a lot more abstractions and expressive power than languages like C and C++
  - This tends to come at a cost of performance, though
  - We'll be looking at Python specifically for this lecture

What is Python?

The horse's mouth:

- "Python is an interpreted, interactive, object-oriented programming language. It incorporates modules, exceptions, dynamic typing, very high level dynamic data types, and classes. Python combines remarkable power with very clear syntax."
  - I find the second statement to be very true: it's really easy to do really powerful stuff that reads well and isn't bogged down by weird syntax (*cough* C++ *cough*)
  - One of my favorite languages...coming from a person whose favorite languages include C, assembly languages, and (System)Verilog

What is Python?

- Currently in version 3 (version 2 is at its end-of-life)
- **This lecture is going to focus on Python 3**
- Has an extensive, powerful, easy to use standard library
- Great to use when you want to do something more complicated than can be (easily) handled in a shell script
- Can be used anywhere from data processing to scientific computing to webapps (e.g. Flask) to games (Ren'Py, anyone?)
  - I've used Python for random scripts, autograders, data processing, managing a GitLab server, prototyping a OpenCV app, and working on a (crappy) visual novel

Running Python

- There are multiple ways to run and use Python
  - As a script
  - In its interpreter's shell
  - In an IDE (e.g. Spyder)
- Your system may link/alias the `python` command to `python2` or `python3`
  - Be aware of which one it is: running `$ python --version` can help out
- Script files can be run via the `python`/`python3` command or directly with a shebang (`#!/usr/bin/env python3`)
  - `$ python script.py`
  - `$ ./script.py` (after `chmod`)
- You can run the interactive shell via `$ python`/`$ python3`
  - Good for tinkering with some Python wizardry
- I'm focusing more on its use as a script, but I will use the interactive shell for some demonstrations

Fundamentals

- You all have learned at least one (typed) programming language by now, so I'm going to focus on the parts that make Python "Python"
    - This is going to skim over the basic stuff that every language has (e.g. control flow)
    - Once you learn one language, picking up another language isn't *too* difficult: it's just learning the particular syntax and language quirks
- The source of all this information is the [official Python 3 documentation]() and [its tutorial]()
    - I'm not here to exhaustively just dump reference info onto you: you can easily find the exact behavior of `sequence[i:j]` by perusing the documentation
    - I'm also not here to give you a nice easy step-by-step tutorial on Python: you already know how to write code and the tutorial above and countless others on the internet can get you started.
    - I'm here to highlight the key ideas and features powering Python as a means to both expand and apply your theoretical CS knowledge
    - (By the way, perusing the documentation is how I'm coming up with these slides)

A taste of Python

```
#!/usr/bin/env python3
class Foo:
    def __init__(self, str, num):
        self.x = str
        self.y = num
    def __str__(self):
        return self.x + ": " + str(self.y)

def fib(n):
    seq = [0, 1]
    while len(seq) < n:
        seq.append(seq[len(seq)-1] + seq[len(seq)-2])
    return seq

fibseq = fib(10)
bar = []
for n in fibseq:
    bar.append(Foo('fib', n))
for b in bar:
    print(b)
```

Basics

- Conceptually works much like a shell script interpreter
- Things like functions (and classes) can be entered in manually at the shell, much like with Bash
- Pretty much everything you can do in a script can be done manually at the shell, so if you wanted to play around with stuff you could do that
- Semicolons not required; they can be used to put multiple statements on a single line
- Meaningful whitespace
  - Instead of using keywords like `do` and `done` or things like curly brackets, indentations are used to mark the scope of code blocks

Variables and Data

- Understanding how Python handles data is essential to understanding Python
- Info comes from the [Data model section] by the way
- Every datum is an *object* (this includes functions!)
- Every object consists of an ID, type, and value
    - Value also consists of *attributes* (i.e. member variables)
- The type determines *mutability*
    - *Mutable* objects have values that can change
    - *Immutable* objects have values that can't change
- A *variable* is a **reference** to a particular *object*
    - Variables can be assigned via `=`
    - Assignment really mean that it becomes a reference to the RHS's object
- `id(var)` and `type(var)` will return the ID and type of the object referenced by variable `var`

Playing with variables and objects

```
a = 5 # "a" becomes a reference to an integer whose value is "5"
b = a # "b" becomes a reference to the object "a" refers to
print(id(a))
print(id(b))
print(a is b)
b = 7          # ?
print(id(b))   # ?
print(a is b) # ?
```

When we look at the built-in types we'll why this happens

Built-in types (the important ones)

- Type info comes from its section in Data model
- Literal info comes from its section in Lexical Analysis for you programming languages (PL)/compilers nerds
- There's a bunch of built-in functions and operations that they can do: refer to the standard library reference manual for details.

None

- Indicates "lack" of value; analogous to null
- `None`
- Functions that don't return anything return `None`

Numbers

- These are **immutable**! A new number is a new object!
  - Think about how this affected the behavior in the previous example
- `int`: represent integers
  - Literals: `12345`, `0b01001101`, `0o664`, `0xbaadf00d`
  - (As of 3.6 you can also insert `_` to group digits to make long literals more readable e.g. `0b0100_1101`)
- `bool`: special integers that represent truth values
  - Values can be `True` (1) and `False` (0)
- `float`: double-precision floating point
  - Literals: `12345.0`, `12345.`, `1e10`, `1e-10`, `1e+10`
- `complex`: pair of double-precision floating point numbers
  - `real` and `imag` components
  - Imaginary literals: like regular float literals but with a `j` after e.g. `12345.0j`

Sequences

- *Ordered* "sets" (think "array") that are indexable via `[]`

Mutable sequences

- Lists (`list`)
  - Sequence of arbitrary objects (like a Tuple but mutable)
  - Created via a comma-delimited list of expressions in square brackets e.g. `[1,2,3,4,5]`, `[]`
- Byte arrays (`bytearray`)
  - Sequence of 8-bit bytes (like a Bytes but mutable)
  - Created via the `bytearray()` function

Immutable sequences

- Strings (`str`)
  - Sequence of *Unicode code points* from `U+0000 - U+10FFF`; this means that each character isn't necessarily a byte!
  - Literals: `'string contents'` and `"string contents"`
  - `encode()` can convert a string into raw bytes given an encoding
- Bytes (`bytes`)
  - Sequences of 8-bit bytes (like a Bytearray but immutable)
  - Literal: `b'some ASCII string'`, `b"some ASCII string"`
  - `decode()` can convert a bytes object into a String given an encoding

Immutable sequences

- Tuples (`tuple`)
  - Sequence of arbitrary objects (like a List but immutable)
  - Created via a comma-delimited list of expressions e.q. `1,2,3,4,5`
  - You can wrap it in parentheses to separate it from other stuff e.g. `(1,2,3,4,5)`
  - Note that it's the commas that make tuples: there's an exception where an empty tuple is created by `()`
  - This is the magic behind the returning of "multiple objects" and "multiple assignment" e.g. `a,b,c = 1,2,3`

Sets

- Unordered sets of *unique, immutable* objects
- Sets: mutable sets (`set`)
  - Created via the `set()` function or comma-delimited list of expressions with curly brackets
  - `{1, 2, 3, 4}`
- Frozen sets: immutable sets (`frozenset`)
  - Created via the `frozenset()` function

Mappings

- "These represent finite sets of objects indexed by arbitrary index sets"
    - i.e. they're maps/associative arrays etc.
    - Stores key-value pairs
- Only one type (right now): Dictionaries (`dict`)
    - Mutable
    - Created via `{}`: e.g. `{ key1:value1, key2:value2 }`
    - Keys can be of any immutable, hashable type
    - Indexable via key: e.g. `some_dict[some_key]`, `another_dict['string key']`
    - Add items by indexing via some key: e.g. `some_dict['hello'] = 'world'` will add the pair `'hello':'world'` to the dictionary

Callables

- Yes, functions themselves are objects with particular types
- This means that you can easily assign variables to them!

```
p = print
p('hello world!')
```

Some callable types (there's more as well)

- Each of these have special attributes that describe some component of it e.g. `__defaults__`, `__code__`
- User-defined functions
- Instance methods (i.e. class member functions)
  - The `__self__` attribute refers to the class instance object and gets implicitly passed as the leftmost argument
  - `some_instance.some_func()`
- Classes
  - Yes, these are callable: by default they produce new object instances when called
  - `some_instance = MyClass(some_arg)`

Expressions

- There's a lot of nitty-gritty details in the [manual]() if you're interested
- These are units of text that resolve into some sort of value
- Identifier: `varname`
- Literal: `123`, `'some string'`, `b'some bytes'`
- Enclosure: `(123 + 23)`, `['i', 'am', 'a', 'list']`, `{1:'dict', 2:'view'}`
- Attribute reference (i.e. member access): `.`
  - e.g. `someobject.someattr`

Expressions

- Subscription: `[<index>]`
  - Implemented by things like sequences and dictionaries
- Slicing: `[lower:upper:stride]`
  - e.g. `somelist[1:3]`
  - A selection of items in a sequence
  - Multiple ways to specify one
- Calls: `foo(arg1, arg2)`
  - For callable objects, which include functions/classes

Operators (some can be implemented/overloaded!)

- Power: `**`
  - `2 ** 5`: "2 to the power of 5"
- Unary: `-`, `+`, `~`
  - `-2`
- Binary arithmetic: `+`, `-`, `*`, `/`, `//`, `%`, `@`
  - `/` is a real division, `//` is a floor division (i.e. integer division)
  - `@` is intended for matrix multiplication, but no built-ins implement it
- Binary bitwise: `&`, `|`, `^`
  - `0x5a5a | 0xa5a5`
- Shifting: `<<`, `>>`
  - `1 << 5`

Operators (some can be implemented/overloaded!)

- Comparison: `<`, `>`, `==`, `>=`, `<=`, `!=`, `is`, `is not`
  - `a == b`, `a is b`
- Membership: `in`, `not in`
  - `i in [0, 1, 2, 3]`
- Boolean: `not`, `and`, `or`
  - `a and b`, `a or b`, `not a`
- Conditional/ternary: `x if C else y` (analogous to C/C++ `C ? x : y`)
  - If `C` is `True`, evaluates `x`, else evaluates `y`

Comprehensions

- "Pythonic" way to create lists, sets, and dictionaries
- Iterates over an iterable object allowing you to perform operations
- Optional conditional to filter out certain objects
- List comprehension
  - `[s.name for s in students]`
  - `[s.name for s in students if s.grade > 70]`
- Set comprehension
  - `{s.name[0] for s in students]}`
  - `{s.name[0] for s in students if s.grade > 70]}`
- Dictionary comprehension
  - `{s.name:s.grade for s in students}`
  - `{s.name:s.grade for s in students if s.name[0] == 'A'}`
- There's more to them, like multiple `for` and `if`
  - Check out the [tutorial]() and the [reference manual]()

Simple statements (some of them)

- Simple statements are statements that are on one line
  - You can put multiple simple statements on one line by separating them with semicolons
- The examples are not exhaustive: for instance, there's many different kinds of exceptions that can be raised
- Expressions
  - `a` (for some variable `a`)
  - `5 + 3`
  - `foo()`
  - The object the expression resolves to will be printed out at the interactive shell

- Assignments: bind a variable to some object (or one produced by an expression)
  - `a = 5`
  - `b = 'hello'`
- Augmented assignments: combine binary operation and assignment
  - `a += 1`
- `assert`: assertion
  - `assert a > 0`
- `del`: deletes
  - Can unbinds variable(s); various classes can overload this for different behaviors
  - `del a`
  - `del sequence[3]`

- `return`: leaves a function call
  - Can just return `return`
  - Can specify an object to return `return a`
  - Can return "multiple" objects inside a *tuple* `return a,b,c`
- `pass`: no-op, used where a statement is needed but you don't want to do anything
- `raise`: raises an exception
  - `raise Exception("oops")`
- `break`: break out of a loop
- `continue`: skips the rest of current iteration of a loop and go to the next
- `import`: imports a module; more on this later

Compound statements

- *Compound statements* are called so as they group multiple statements
- You've got your standard bevy of control flow elements as well as try-catch and functions and classes
- Composed of a *header* (keyword and ends in colon e.g. `def hello():`) and a *suite* (the stuff "inside")
- The suite is a code block, which is either on the same line of the header or indented on the following lines

```
def function1(arg): # this is the "header"
    pass # these statements
    pass # are in the suite

def function2(arg): pass; pass; pass; # suite on the same line
```

if-elif-else

```python
if a > b:
    print('a > b')
elif a < b:
    print('a < b')
else:
    print('a == b')
```

while

```python
while a > b:
    print(a)
    a -= 1
```

for

- Iterates over an iterable object such as a sequence (e.g. list, string)

```python
list = ['hello', 'world', 'foo', 'bar']
for x in list:
    print(x)

# range() is a built-in function that returns an
# immutable iterable sequence of integers
for i in range(len(list)):
    print(list[i])
```

`try`

- Allows you to handle exceptions and perform cleanup

```
# a = 1
a = 0
try:
    b = 5 // a
except ZeroDivisionError:
    print("oopsie")
finally:
    print("cleanup...")
```

`with`

- This one is a bit more complicated: it adds some convenience factor to `try`-`except`-`finally`
  - Details in the reference manual!
  - In short, there's special functions tied to certain objects that will automatically get called when exceptions get raised
- You see this a lot when opening files, where it can close files for you without your explicitly calling `close()`

`with`

```python
with open("somefile.txt", "r") as f:
    data = f.read()

# *similar* to, not *equivalent*
# the equivalent is a bit more complex
hit_except = False
try:
    f = open("somefile.txt", "r")
except:
    hit_except = True
finally:
    if not hit_except:
        f.close()
```

Functions and classes

- The definitions are compound statements
- I put them in their own section because they also have a usage component

Functions

- Fairly self explanatory, with a neat feature of optional arguments
- Terminology for calling:
  - Positional argument: "typical", specified by order of your arguments
  - Keyword argument: specified by the name of the argument
  - Default argument: definition provides a default value

```python
def func1():
    pass # hey, a use for pass!

def func2(arg1, arg2="default"):
    print(arg1 + " " + arg2)

def func3(arg1, arg2="default", arg3="default"):
    print(arg1 + " " + arg2 + " " + arg3)

func1()

func2("arg1") # arg2 defaults to "default"
func2("arg1", "arg2") # use of positional arguments

func3("arg1", arg3="arg3") # use of keyword argument
```

Classes

- Also fairly self explanatory
- Class definitions really just customize class objects
- Classes have special functions that you can implement things like "constructors" and do the equivalent of operator overloading from C++
- Remember that classes are *callable*: when called they run their `__new()__` function to make a new instance, and then by default pass the arguments to the instance's `__init()__`
- (These `__xxx()__` functions are called "dunder" methods and serve as a way to implement underlying behavior for various things e.g. operator "overloading")

```python
class Foo:
    # variables here are class attributes: they're analogous
    # to static class variables in other languages
    num_foos = 0

    # you can define functions inside of a class definition
    # that will become your member functions ("methods")

    # __init__() is like a constructor
    # The first argument is a special variable that refers to
    # the instance, analogous to "this" in C++, but is implicit
    def __init__(self, arg1, arg2, arg3):
        # this is where we set member variables of class instances
        self.a = arg1
        self.b = arg2
        self.c = arg3
        type(self).num_foos += 1

    def somefunc(self):
        return self.a + self.b + self.c

foo_instance = Foo('a', 'b', 'c')
print(foo_instance.somefunc())
print(Foo.num_foos)
```

An example of "operator overloading"

```
class Foo:
    num_foos = 0

    def __init__(self, arg1, arg2, arg3):
        self.a = arg1
        self.b = arg2
        self.c = arg3
        type(self).num_foos += 1

    # "overload" the + operator
    def __add__(self, other):
        if type(other) is Foo:
            return Foo(self.a + other.a,
                       self.b + other.b,
                       self.c + other.c)
        return None

    def somefunc(self):
        return self.a + self.b + self.c

foo1 = Foo('a', 'b', 'c')
foo2 = Foo('d', 'e', 'f')
print((foo1 + foo2).somefunc())
```

Modules and packages and the standard library

- So far we've gone over things that are built directly into the Python language itself
- Python also comes with an extensive standard library that can do lots of stuff from common mathematical operations to networking
- The standard library has a [detailed manual]()
    - Details not just standard library stuff but also the built-in functions and operations that can be done on the built-in types

Importing

- To make use of the standard library, you'll have to `import` the modules
  - `import sys` will import the `sys` module
  - `import math` will import the `math` module
- This will make the things defined in the module accessible through some identifier, which by default is the module's name
  - `sys.argv` accesses the script's argument list, which is under the `sys` module
- You can also have `import` use another identifier for that module
  - `import sys as s` will allow you to identify the `sys` module as `s`
  - `import tensorflow as tf`

What is a module anyway?

- A module is a unit of Python code
  - A module can comprise of a single or multiple files
- In a directory with `some_module.py` and `user.py`, `user.py` could have:

```
import some_module

some_module.cool_thing()
```

- The `import` process will search a predefined search path and then the current directory

Then what's a package?

- A Python package is a special kind of module that has a sort of hierarchy of subpackages e.g. `email.mime.text`, where `email` is a package that has a subpackage `mime`

Package managers

- You're not restricted to just the standard library and your own modules
- You can also install modules and packages used by other people
  - NumPy, Matplotlib, SciPy, OpenCV to name a few
- The two most common ones are `pip` and `conda` (associated with the Anaconda distribution of Python)
  - Sometimes a particular Linux distribution's package manager will also manage Python packages e.g. `pacman`

Useful tidbits

Built-ins

I/O

- `print()`
- `open()`

Types

- `len(sequence)` will get the length of a sequence
- `str(obj)` to get a string representation of an object
- `int(obj)` produce an integer from a string or other number
- `list.append()` (and its friends) to manipulate lists
- `range()` to produce a `range` object, which is an immutable sequence of numbers
  - Useful for `for` loops
- `dict.values()` provides an iterable object with the values of a `dict` (dictionary)

Standard library modules

- `sys`, `os`, `io`, `math`, `statistics`, `copy`, `csv`, `re`
- A lot of the other ones are application dependent

Library functions and attributes

- `sys.argv`: list of command-line arguments
- `os.system("ls -a")`: run a shell command
- `subprocess.run(['ls', '-l'], capture_output=True).stdout.decode('utf-8')`: run a shell command, get its output, decode to string via UTF-8
- `copy.copy()`: perform a shallow copy of an object
- `copy.deepcopy()`: perform a deep copy of an object
- `math.ceil()`, `math.floor()`
- [`read()`](), [`write()`](), [`close()`]()
  - Depending on how you `open()` a file, you'll get different file object types (e.g. text vs binary) with different attributes

Looking back at our taste of Python

```python
#!/usr/bin/env python3
class Foo:
    def __init__(self, str, num):
        self.x = str
        self.y = num
    def __str__(self):
        return self.x + ": " + str(self.y)

def fib(n):
    seq = [0, 1]
    while len(seq) < n:
        seq.append(seq[len(seq)-1] + seq[len(seq)-2])
    return seq

fibseq = fib(10)
bar = []
for n in fibseq:
    bar.append(Foo('fib', n))
for b in bar:
    print(b)
```

Extra

A bit out of the scope of this one lecture, but useful things to look at

Perhaps these will be advanced exercises 🤔

Debugging with `pdb`

- Standard library module that provides debugging support
- [Reference manual entry]

NumPy

- Package that provides fundamental types and operations for scientific applications
- Well known for its array type
  - Also has useful functions such as FFTs
  - These are optimized for performance!
  - NumPy arrays serve as one of the backbones of Python-based scientific computation
- User guide

SciPy

- Package that provides functions and algorithms for scientific computation
  - Linear algebra, FFTs, stats etc.
- Refence

Matplotlib

- Package that provides visualization functions for making graphs and stuff
- User guide

With NumPy, and SciPy, Matplotlib, who needs MATLAB?

```
Not a fan of it as a language (also $$$), but its libraries and utilities a
```

Questions?