Python Introduction

Principles of Programming Languages
Colorado School of Mines

https://lambda.mines.edu
Why Python?

Why study Python in Principles of Programming Languages?

- Multi-paradigm
  - Object-oriented
  - Functional
  - Procedural
- Dynamically typed
- Relatively simple with little feature multiplicity
- Readability focused
- No specialized IDE required
- Fast, relative to other dynamically typed languages
  - And when it’s not fast enough, you can rewrite that performance-critical section in C. Python is natural to interop with C.
- Highly General Purpose!
  - Web programming, machine learning, GUI programming, Email processing, education, simulations, web scraping...
Installing Python

For this course, we will be using **Python 3.6 or 3.7**.

- **ALAMODE machines**: already have Python 3.7
- **Arch Linux**: install python for 3.7
- **Ubuntu 18.04**: install the python3 package for 3.6
- **Ubuntu 16.04 or 14.04**: setup the ppa:deadsnakes/ppa then install python3.7
- **Fedora 28**: ships with Python 3.6
- **Other distros**: ask on Piazza if you need help

**Note**

You are *required* to develop on Linux. I am unable to provide help for you setting up the projects on other systems.
Python is one of the few languages with an official style guide (PEP 8). Here’s a quick summary:

- Use 4-spaces for each level of indentation. **Never use hard tabs!**
- Use snake_case for function and variable names.
- Use CapWords for class names.
- Never ever use camelCase in Python.
Basic Input and Output

- The `print` function takes any amount of arguments, and prints them separated by spaces on the same line.
- The `input` function takes an optional prompt string, prompts the user for input, and returns the string they typed.

```python
name = input("What is your name? ")
print("Nice to meet you", name)
```
A Simple Example

```python
for i in range(1, 101):
    if i % 3 == 0 and i % 5 == 0:
        print("Fizz Buzz")
    elif i % 3 == 0:
        print("Fizz")
    elif i % 5 == 0:
        print("Buzz")
    else:
        print(i)
```

Indentation Denotes Scope

Any time Python sees a :, it expects an indented section to follow. The indented section denotes the scope of the operation.
Builtin Types

bool: True or False
int: integers, not size-bound
float: double-precision floating point numbers
complex: complex numbers
str: for Unicode strings, immutable
bytes: for a sequence of bytes, immutable
tuple: immutable ordered storage
set: mutable unordered storage
frozenset: immutable unordered storage
dict: mutable key-value relation

Functions: yup, they’re first class!
Classes: they’re first class too (of type type)
# List literals
[1, 2, 3]

# Tuple literals
(1, 2, 3)

# ... 1 element tuples are special
(1,)

# Dictionary literals
{'Ada': 'Lovelace', 'Alan': 'Turing'}

# Set literals
{1, 2, 3}

# ...empty set is:
set()
String Formatting

To format elements into a string, you could convert each element to a string then add them all together:

```python
print("Time " + str(hours) + "::" + str(minutes) + ".")
```

Ow... my fingers hurt, and that was not too easy to read either. As an alternative, try .format on a string:

```python
print("Time {}::{}.format(hours, minutes)
```

Or, since Python 3.6, you can use an f-string:

```python
print(f"Time {hours}:{minutes}.")
```

See the Python documentation for more information. There’s plenty to this formatting language.

**Note**

Do not use old-style (printf-style) string formatting in this course.
Selection (if statements)

Python’s primary structure for selection is if:

```python
if i == 0 and j == 1:
    print(i, j)
elif i > 10 or j < 0:
    print("whoa!")
else:
    print("all is fine")
```

Notice you do not need parentheses surrounding the condition like in C or C++.

There’s also a ternary operator (good for simple conditionals):

```python
def foo(bar, baz):
    return bar if bar else baz
```
Why no switch or case?

Most switch or case statements over-complicate what could be done in a single line using a dictionary. Where this is not the case, you really shouldn’t be using a switch anyway.

An Example switch in C

```c
switch (c) {
    case 'q':
        a++;
        break;
    case 'x':
        a--; 
        break;
    case 'z':
        a += 4;
}
```

Python Equivalent

```python
diff = {'q': 1, 'x': -1, 'z': 4}
a += diff[c]
```
Python provides your traditional `while` loop, the syntax is similar to `if`:

```python
while n < 100:
    j /= n
    n += j
```

But under most cases, the **range-based** `for` loop is preferred:

```python
for x in mylist:
    print(x)
```

**Note**

Python’s `for` loop is a range-based `for` loop, unlike C’s `for` loop which is really just a fancy `while` loop.
Generating Ranges

The generator function range creates an iterable for looping over a sequence of numbers. The syntax is `range(start, stop, step).

- start is the number to start on
- stop is the number to stop before
- step is the amount to increment each time

```python
for i in range(0, 5, 1):
    print(i)
```

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
</table>

Optional Parameters

Both start and step are optional, and if omitted, will be assumed to be 0 and 1 respectively.
In Python, you can pair an else block with for and while. The block will be executed only if the loop finishes without encountering a break statement.

An example of this can be seen below:

```
for i in range(10):
    x = input("Enter your guess: ")
    if i == x:
        print("You win!")
        break
    else:
        print("Truly incompetent!")
```
mylist = [1, 2, 3, 4]

# syntax is [start:stop:step], step optional
mylist[1:3] # => [2, 3]

# unused parameters can be omitted
mylist[::1] # => [4, 3, 2, 1]

# without the first element
mylist[1:] # => [2, 3, 4]

# without the last element
mylist[::1] # => [1, 2, 3]
Multiple assignments work like so:

```python
tuples = ("R. Stallman", "L. Torvalds", "B. Joy")
a, b, c = tuples
```

* can be used to collect a tuple:

```python
# drop the lowest and highest grade
grades = (79, 81, 93, 95, 99)
lowest, *grades, highest = grades
```

The same can be done to expand a tuple in a function call:

```python
# Each grade becomes a separate argument
print(*grades)
```
To define a function in Python, use the `def` syntax:

```python
def myfun(arg1, arg2, arg3):
    if arg1 == 'hello':
        return arg2
    return arg3
```

Even if your function does not take arguments, you still need the parentheses:

```python
def noargs():
    print("I'm all lonely without arguments...")
```
Keyword Arguments

When we define a function in Python we may define **keyword arguments**. Keyword arguments differ from **positional arguments** in that keyword arguments:

- Take a default value if unspecified
- Can be placed either in order or out of order:
  - **In order**: arguments are assigned in the order of the function definition
  - **Out of order**: the argument name is written in the call
- Positional and keyword arguments can be mixed, so as long as the positional arguments go first.
Keyword Arguments: Example

```python
def point_twister(x, y=1, z=0):
    return x + 2*z - y
```

# all of these are valid calls
print(point_twister(1, 2, 3))  # x=1, y=2, z=3
print(point_twister(1, 2))    # x=1, y=2, z=0
print(point_twister(1))        # x=1, y=1, z=0
print(point_twister(1, z=2, y=0))  # x=1, y=0, z=2
print(point_twister(1, z=2))   # x=1, y=1, z=2

Style Note

PEP 8 says that we should place spaces around our "=" in assignments, but these are not assignments, and should be written without spaces around the "=".
Just like a tuple or list can be expanded to the positional arguments of a function call using *some_tuple, a dictionary can be expanded to the keyword arguments of a function using **some_dict. For example:

```python
my_point = {'x': 10, 'y': 15, 'z': 20}
print(point_twister(**my_point))
```
Python allows you to define functions that take a variable number of positional (*args) or keyword (**kwargs) arguments. In principle, this really just works like tuple expansion/collection.

```python
def crazyprinter(*args, **kwargs):
    for arg in args:
        print(arg)
    for k, v in kwargs.items():
        print('{}={}'.format(k, v))

crazyprinter("hello", "cheese", bar="foo")
# hello
# cheese
# bar=foo
```

The names args and kwargs are merely a convention. For example, you could use the names rest and kwds instead if you wanted.
*args and **kwargs: Another Example

```python
def fancy_args(a, b, *args, c=10, **kwargs):
    print("a is", a)
    print("b is", b)
    print("c is", c)
    print("args is", args)
    print("kwargs is", kwargs)

fancy_args(1, 2, 3, 4, c=15, d=16, e=17)
# a is 1
# b is 2
# c is 15
# args is (3, 4)
# kwargs is {'d': 16, 'e': 17}
```