

Python Introduction

Principles of Programming Languages

Colorado School of Mines

<https://lambda.mines.edu>

Learning Group Activity

Review the learning group activity with your group members:

- 1 Which activity did you choose?
- 2 What problems did you encounter, if any? Can your group members help?
- 3 What did you learn from the activity?

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**.

Windows and Mac OS X:

Install from <https://www.python.org/downloads>

Linux:

Check your distribution's package manager for Python 3. If you are using a distribution that only has Python 3.5 (e.g., Ubuntu 16.04), that should be fine. Just don't go any older than 3.5.

Note

While you are free to develop on whatever system you choose, your code is expected to work on a Linux machine. Therefore, it is recommended that you setup Linux on your computer.

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.

For your LGA, you will get to explore the style guide in further detail.

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.

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

A Simple Example

```
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
- list*: mutable ordered storage
- 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)

Literals

```
# 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:

```
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:

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

Or, if you have Python 3.6, use an f-string:

```
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`:

```
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):

```
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

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

Python Equivalent

```
diff = {'q': 1, 'x': -1, 'z': 4}  
a += diff[c]
```

Iteration

Python provides your traditional while loop, the syntax is similar to if:

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

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

```
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

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

```
0  
1  
2  
3  
4
```

Optional Parameters

Both `start` and `step` are optional, and if omitted, will be assumed to be `0` and `1` respectively.

Pairing Iteration Structures with else

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!")
```

Slicing

```
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]
```


Tuple Expansion & Collection

Multiple assignments work like so:

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

* can be used to collect a tuple:

```
# 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:

```
# Each grade becomes a separate argument  
print(*grades)
```

Functions

To define a function in Python, use the def syntax:

```
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:

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

*args and **kwargs

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.

```
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
```

- Modules
- Generator Functions
- List Comprehensions
- Functional Programming