Lab 1
Python Review

COMS 2132 - Professor Bauer

Dynamically Typed

- Python is a dynamically typed programming language, meaning that python does not know the type of a variable until the **code is run**
- Therefore, you *do not have to declare the type* of variables while coding with Python!

int x = 50

String x = "Hello World!"
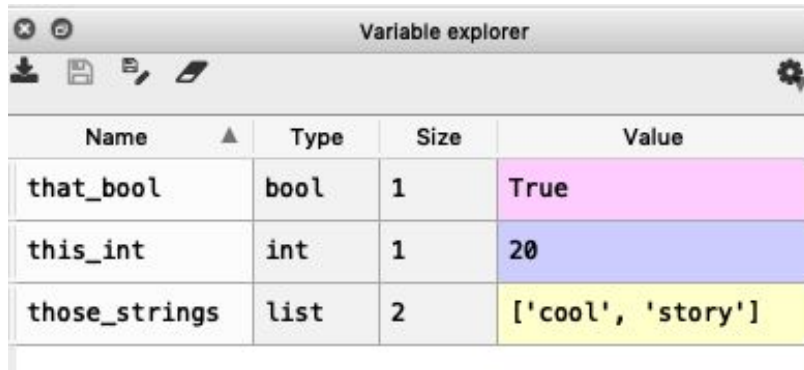
x = 50

x = "Hello World!"

Variables

- Variables contain different **objects** within a program
- Variables can be retrieved, stored, and modified after they are created
- Variables are assigned using a single **=**
    - this_var = 3.14
- Some data types we have stored in variables thus far:
    - **int**   1   0
    - **float**   3.0   2.1
    - **str**   'thing'   '3.1'
    - **list**   ['a', 'b', 'c']   [True, False]
   - **bool**   True   False

The Namespace

- Any time a variable or function is created, it is stored in the iPython console's namespace
- The namespace maps the different variable names to their associated values

Name	Type	Size	Value
that_bool	bool	1	True
this_int	int	1	20
those_strings	list	2	['cool', 'story']

variables   values

this_int ➝ 20

that_bool ➝ True

those_strings ➝ ['cool', 'story']

namespace
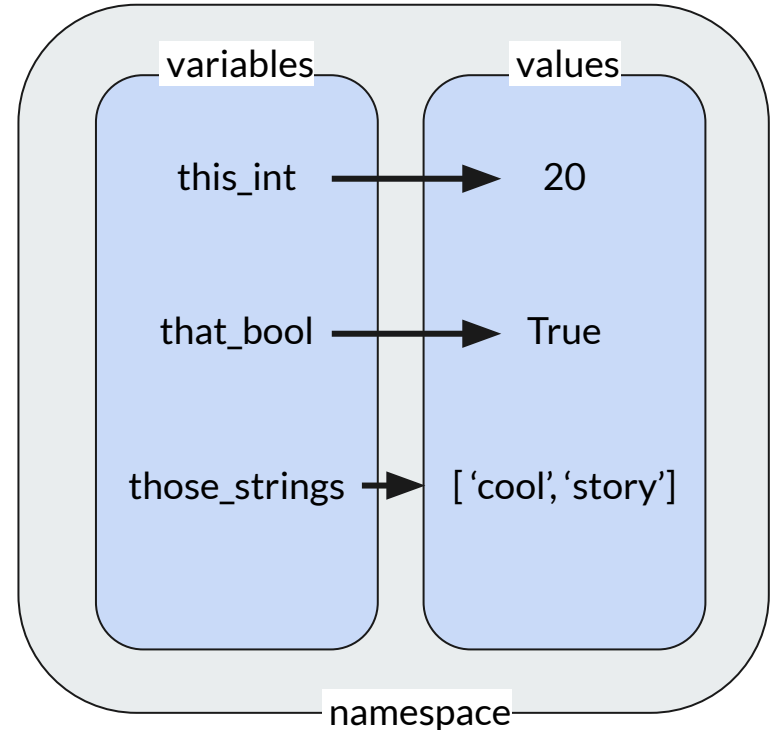
Objects and Types

- *Everything* we assign to a variable is an object!
    - Remember, variables have **no type.**
- Every object has its own identity, attributes, and name(s)
    - Just because two objects *look* the same, it does not mean that their variables point to the same objects (their individual IDs may be different)
    - Likewise, names can refer to different objects at different times and more than one name can refer to one object
- Comparing objects can return True or False
    - Saying "is" will compare the IDs of two objects
    - Saying "==" will say "are these equal?"

Expressions and Statements

- **Expressions**   **this == that**
    - Get evaluated and **return** a value
    - Calling an expression in the iPython console will show the value returned
    - Returned values can be caught and stored in variables
- **Statements**
    - Do not return a value
    - Simply **states** something   **this = 4**
    - Can have side effects

Operators

- Numeric data types are compatible with many operators
- Most look familiar (+ , - , * , /)
- Some not so much:
    - //   forced integer division   7 // 3 = 2
    - %   modulo   7 % 3 = 1
    - **   exponent   2 ** 3 = 8
- Operators can mean different things for different data types
    - Ex: + used on strings is not the same as + on ints
- Some operators are not applicable to some data types -- be mindful of errors!

Boolean Expressions

- **Boolean** expressions, hence the name, are a useful and popular expression
    - They utilize operators to **return** either **True** or **False**
- Boolean expressions have a few operators themselves
    - **and**   only returns true if both items are true
    - **or**   returns true if at least one item is true
    - **not**   returns the opposite of the following truth value

A	B	A and B
T	T	T
T	F	F
F	T	F
F	F	F

A	B	A or B
T	T	T
T	F	T
F	T	T
F	F	F

A	not A
T	F
F	T

If, elif, and else

- **If** statements cause boolean **expressions** to be evaluated, causing the below block of code to be run only **if** it returns true
    - Applies **selection**
- **Elif** is the same as an if statement, but will only be checked if the if statement/elif statement before it returns false
- **Else** is a catch-all, it will only run the below block of code if none of the elifs/ifs are run
- Important use: **if _name_ == "_main_":**

```
this = __
that = __
if this == 2:
    print('one')
elif this ==1 or that == 4:
    print('two')
else:
    print('three')
```

Loops - Repetition

- **While**
    - Works the same as an if statement, but checks the truth value of the boolean expression at the end of each iteration
    - Know how to utilize a count

while (boolean_expression):
~do some stuff
reevaluates expression

- **For**
    - Works its way through a pre-defined iterable (a range, a list, a string etc), executing the loop body **for** each element in the iterable

for (thing *in* thing_to_iterate):
~do some stuff

Break and Continue

- **Break**
  - A break statement will terminate the *innermost* loop in which it is present, continuing on with the rest of the code

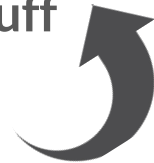**while (boolean_expression):**

~do some stuff

break

- **Continue**
  - Skips the remaining lines of the *innermost* loop of the code, but doesn't terminate the loop -- it reevaluates the boolean expression

while (boolean_expression):

~do some stuff

continue

Lists

- **Lists** can have many **different** data types stored within them, just like a grocery list or a to-do list

<div align="center">this_list = [True, 1, 2.0, "wow", [1, 2 ,3]]</div>

- Lists have **lengths** and **indices** to access a certain object within
  - Indices always **start at 0** and go until the **length of the list - 1**
  - You can **append** to a list (adding an item to the end of the list)
  - You can also **sort** a list with a built-in sort function
  - Lists are **iterable** -- you can work through them with a for loop
- Lists can also contain more lists! Be careful when reading these.
- To copy a list, use a **[:]** in a new list definition.

Strings

- **Strings** can be created with 'apostrophes' or "quotation marks"
- Strings, like lists, can be **indexed** and **iterated** through
- Strings are **immutable** -- when the compiler creates a string, it can never change the original string object
    - It looks like strings can be changed, but in reality new strings are created when edits are made
- You can check for substrings using the **in** operator   'bc' **in** 'abc'
- You can **concatenate** (combine) strings using the **+** operator
    - The **format( )** function can also be used

"This is the {} string in the world".format("most interesting")

Tuples

- **Tuples** are **immutable**, like strings.
- Like lists, tuples can have many data types stored in them.
    - Tuples cannot, however, append new items (immutable).

$$\textbf{this\_tuple} = \textbf{(} \text{True}, 1, 2.0, \text{"wow"}, [1, 2, 3] \textbf{)}$$

- Tuples are often returned from functions to allow for more than one element to be sent as output. They can then be **unpacked** and used for further computation.

Slicing and Indexing

-   Both lists and strings can be **sliced** -- cut into smaller sections

this_is_a_list[a : b : c]

a - the **start** index of the slice (**inclusive**)
b - the **end** index of the slice (**exclusive**)
c - the **step size** of the slice (how much it counts by)

-   To go backwards, make **c negative**
-   The index **-1** is the last element in the string or list.
-   For lists within lists, you index twice, getting more specific from left to right
this_list [index1] [index2]

Functions

- To define a function, use the **def** keyword (literally means define)
- To **call** or **invoke** a function, just use the function's name with parentheses wrapped around any **arguments** (inputs)
  - A function must be defined **before** it is called!
- **Return** is not the same as **print**

```
def this_function( a, b, c)
    if a == b:

        return c
    return a*b+c
```

3 arguments -- we'll refer to them as a, b, and c

Two return statements! This is okay.

This one only gets run if the first one does not run

More on Functions

- Functions are **objects**, so they can be stored in variables, too
    - type(foo) is not the same as type(foo( ))
- Any time the compiler sees a function, it evaluates it **before** continuing
- Return statements **terminate** functions
    - The **first** time a return statement is evaluated, the function is tossed aside and the output is used. Nothing below this return statement is evaluated, similar to a break statement.
    - A function without a return will automatically return **None**
- Functions have their own **namespaces** or scopes
    - Be sure to keep a function's dependencies within its scope

Scope

*the arrows indicate where the functions can *look*. Functions can access **global** variables, but not other function's variables. The outer program cannot access the function's variables unless they are returned

outer program

variables

Function 1

variables

Function 2

variables

Function 3
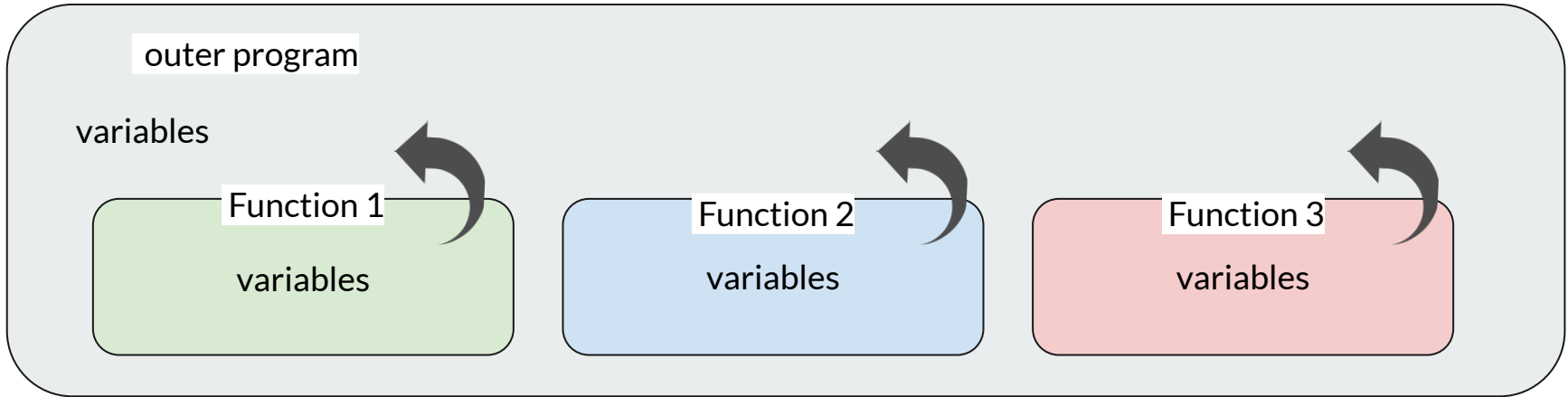
variables

Variables within the functions cannot be seen from outside of the functions.

Variables outside the functions are visible from inside the functions, but it's best not to touch them unless absolutely necessary/intended.

Argument Passing

- When an argument is passed into a function, the original input/variable is left untouched unless it is reassigned

---

```
x = 3
def foo(x):
    return x+1
x = 4
foo(x)
```
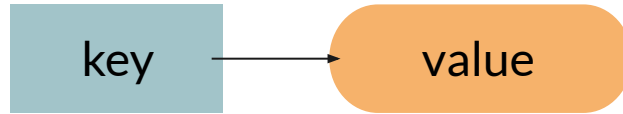True   x == 4
```
x = foo(x)
```
False   x == 4

- Just because x is called the same name, as the x outside of the function, it does not mean they're related.
- Foo is called on the *latest* x, not the x present when foo( ) was defined
- x within scope of outer program is redefined

Dictionaries

| key | → | value |

- Dictionaries act like maps, linking a **key** to a **value**
  - Keys must be **immutable** (string, tuple, integer...)
  - There can be no duplicate keys!
    - Because of this, lookup is extremely fast (constant time)
  - If you know the **key**, you don't need to loop to get the **value**!
- You can create a new dictionary with **{}**

Sets

- **Sets** are similar to lists, but have **no repetition** and have a set of **special operations** that work only on sets
    - union, intersection, difference, superset, subset
    - Sets have no order and cannot be indexed

A = set(1, 2, 3, 4)   B = set(3, 4, 5, 6)

2   3   5

1   4   6
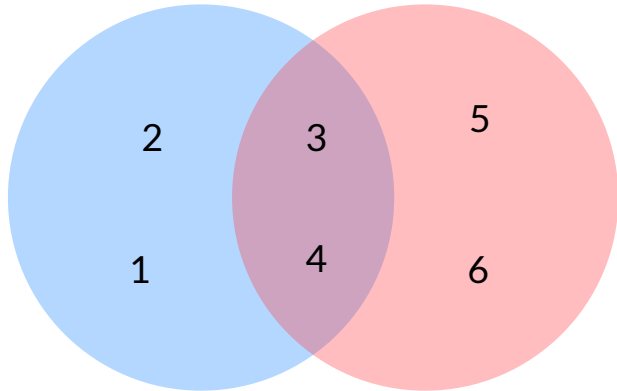
A.union(B) = { 1, 2, 3, 4, 5, 6 }

A.intersection(B) = { 3, 4 }

A.difference(B) = { 1, 2 }

B.difference(A) = { 5, 6 }

A.symmetric_difference(B) =

{ 1, 2, 5, 6 }

A = set(1, 2)   B = set(1, 2, 3, 4, 5, 6)

A is a **subset** of B

B is a **superset** of A

1   5

2

4

3   6
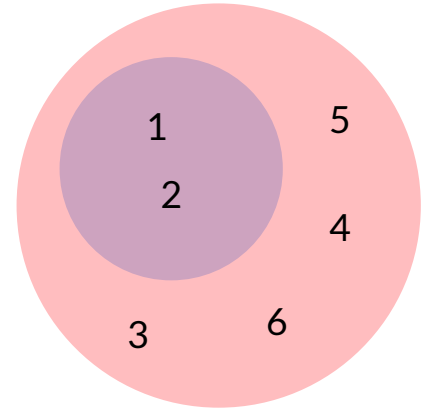
Files

- Files have **extensions** that tell your computer what kind of file it is
  - For binary files, extensions say *how* they are read (.doc, .zip)
  - Text files are already human-readable (.py, .txt)
- You can open a text file using the open function

<p style="text-align:center">file_variable_name = open(A, B)</p>

<p style="text-align:center">A - a string representing the filename/path to be opened</p>

<p style="text-align:center">B - a string representing how the file should be opened</p>

<p style="text-align:center">('r' for read, 'w' for write, 'a' for append)</p>

- A new file is created in 'w' mode. If there is already a file with this name, the previous one is overwritten. Be careful!

File Reading/Writing

- **Newline** characters are where an enter/return key is pressed **(\n)**
- **Tab** characters are when the tab key is entered (~4 spaces) **(\t)**
    - Calling **line.strip( )** in an open file will remove \t or \n from the **end** of the line
- file_variable_name.**read( )** will read the whole file as a string
- file_variable_name.**readline( )** will look at the next line in the file
    - If there is no next line, an empty string is returned by readline( )
- You can print to a writable file by using the typical print statement or using write

> **print("This is what I want to write!" , file=file_variable_name)**
> **file_variable_name.write("I regret what I've written.")**

- file_variable_name.close( ) will close the file (remember how you had to open it?)

Exceptions

- There are two main exception types
    - **Runtime**   errors that occur when a program is running
        - Like malicious user input or edge cases not accounted for
    - **Compile time**   errors that occur before a program can run
        - These are syntax errors, caught when you try to compile a program
- We should catch exceptions by utilizing a **try-except** construct

try:

    ~Try some risky behavior, like trying to open a file that does not exist.

except FileNotFoundError:

    print("It looks like your file could not be found.")

Common errors you will encounter

- **NameError**: if you've never used a variable name before, but try to use a string of characters as if it were a variable name (ex: print(sfasfs))
- **TypeError**: trying to use a function or operation on inappropriate types, for example adding a dictionary plus an integer
- **ValueError**: When a function receives the correct type but a faulty value. For example, trying to convert the string "12.5" into an integer
- **FileNotFoundError**: trying to open up a file that does not exist or cannot be found
- **IndexError**: trying to index past the bounds of a list, index out of range
- **KeyError**: using a key that does not exist to look up a value in a dictionary
- **SyntaxError**: invalid syntax

Recursion

- **Recursion** occurs when a function calls itself
- Recursion requires three conditions
    - A **base case** (where the recursive calls stop)
    - Movement towards this base case (so that an infinite loop does not occur)
    - Faith that the program is working as expected

OOP - Classes and Attributes

- OOP - **Object-Oriented Programming**
  - Three main ideas: **encapsulation**, **polymorphism**, and **inheritance**
- **Classes** are data types that have **a name** and **attributes**
  - these attributes can be **variables, functions/methods,** or **definitions**
  - __str__ , __repr__ , __init__ . . .
- There can be many different **instances** of a class (each instance is an **object**)
  - Each instance has its own **instance variables** - created in the **constructor**
    - Also called data fields -- each object has its own scope
  - The keyword **self** is important to distinguish instance variables (implicitly)

OOP - Encapsulation

? ? ? ? ?

- Data and functionality can be stored within objects
- Methods are ways to **interface** (interact with) the object
  - How interaction actually *works* is unknown to the user
- Think about a thermostat
  - We all know (or can figure out) how to use one
 - Nobody really knows *how* a thermostat makes a room warmer/cooler.

OOP - Polymorphism

- **Polymorphism** is the idea that one operation works on multiple data types
  - The same operation can work on different data types in different ways
  - In OOP, we create polymorphism by writing functions with the same name within different classes (for different objects)

OOP - Inheritance

Parent

Child
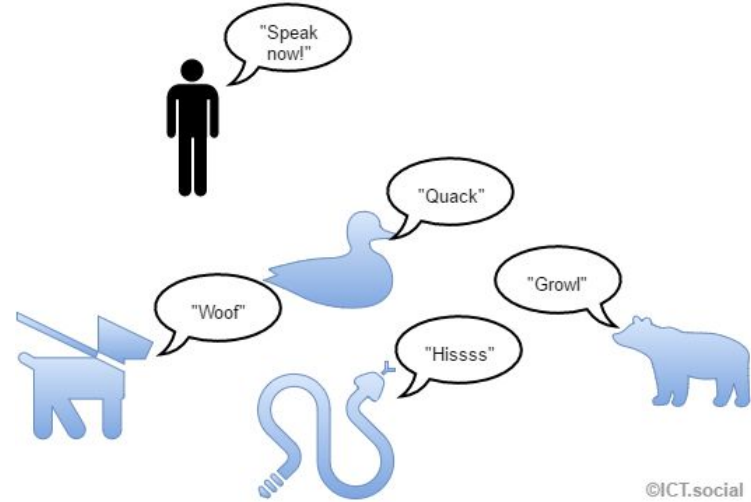
Inheritance

- Classes can piggyback off of other classes, reusing and **inheriting** functions/attributes from them
  - The classes with the original attributes are called **parent classes**
  - The classes that are borrowing/extending the attributes are **child classes**
- You can create a child class by *passing in the parent class as a parameter*
- Use the **super()** function to refer to the parent class and initialize specific attributes/methods
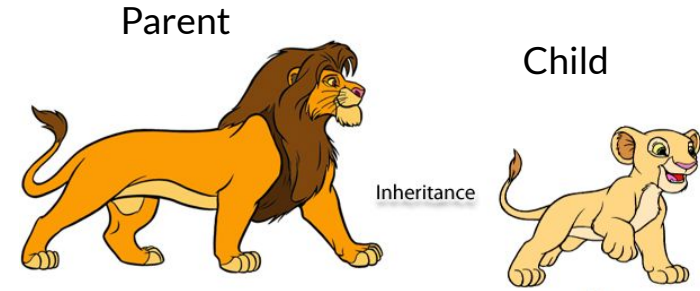
```
class Parent:
    def __init__(self, p_attribute):
        self.p_attribute = p_attribute


class Child(Parent):
    def __init__(self, p_attribute, c_attribute):
        super().__init__(p_attribute)
        self.c_attribute = c_attribute
```

Modules + Packages

- Groups of functions and code can be imported and reused in **modules**
    - Many modules can be combined in **packages**
- Two important packages you might have already used:
    - **Matplotlib** : used for data visualization and plotting of images
    - **Numpy** : used for numeric computing and utilization of matrix arrays

import numpy as np

import random
// x = random.randint(0, 10)

from random import randint
// x = randint(0, 10)

Attendance

- No attendance tracking this week
- Will track attendance next week and onwards! :-)