Python Basics

January 8, 2020

Week 1 of CS 41

Today in CS 41

- Brief Review
- The Data Model
- String Formatting
- File I/O
- Modules
- Virtual Environments

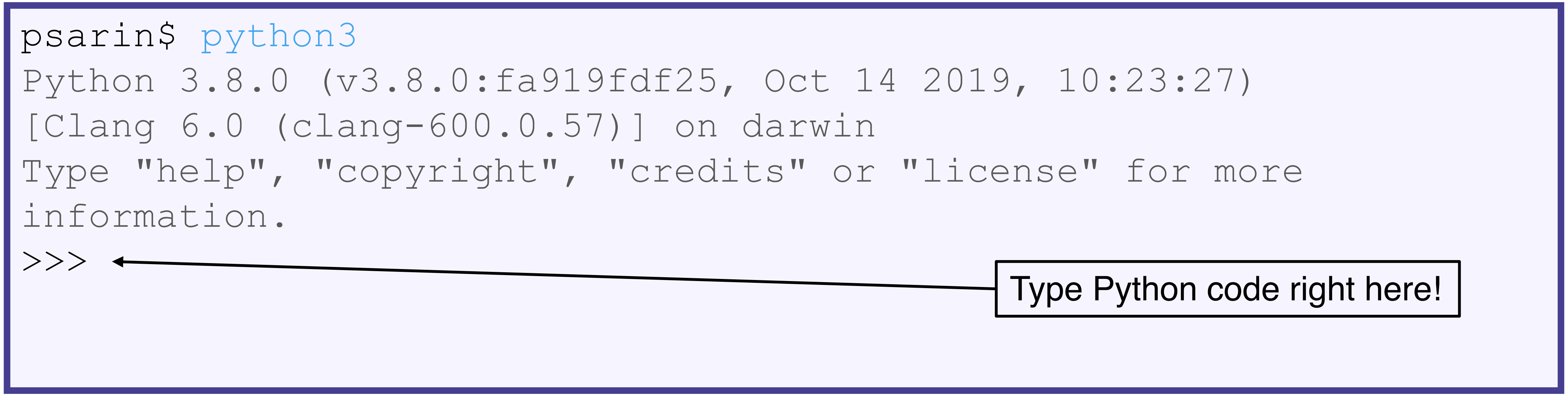
Brief Review

- Interactive interpreter
- Comments
- Variables and types
- Numbers and Booleans
- Strings and lists
- Console I/O
- Control Flow
- Loops
- Functions
- Assignment Expressions

Interactive Interpreter

Python is interpreted, and we can get direct access to its interpreter…

- Run Python code in real-time.



```
psarin$ python3
Python 3.8.0 (v3.8.0:fa919fdf25, Oct 14 2019, 10:23:27)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

Type Python code right here!

Comments

In Python, they start with an octothorpe (pound sign).

```
# Is this thing on?
lecturers = [
    'Michael', # imma let you finish, but...
    ...
]
"""
It's turtles
all
the
way
down.
"""
```

Same as a multiline string!

Variables and Types (so far...)

Python is *dynamically typed*. Variables don’t have a type, but objects do!

```
michael = 22
type(michael) # => int

michael = 'Lecturer, Canadian'
type(michael) # => str
```

Numbers and Booleans

Python has three numeric types: `int`, `float`, and `complex`.

```
5    # => 5 (int)
5.0  # => 5.0 (float)

8_675_309 # => 8675309

3 + 2   # => 5
3 * 2   # => 6
3 ** 2  # => 9
13 / 4  # => 3.25

(3**2 + 4**2) ** (1/2) # => 5.0
```

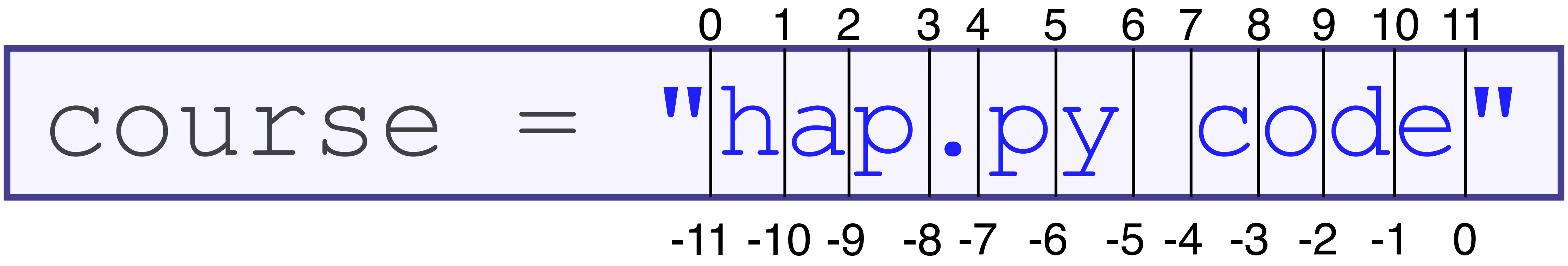
Always a float when the exponent is a float.

Numbers and Booleans

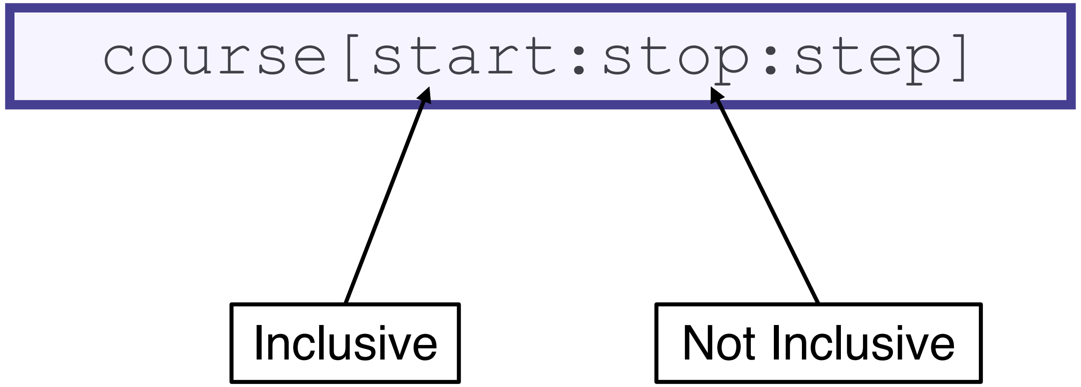
`True` and `False` are sub-types of `int`, with `True == 1` and `False == 0`

```
not True             # => False
True or False        # => True (short-circuits)
True and False       # => False

2 + 3 == 5           # => True
2 + 3 != 5           # => False
1 < 2 < 3            # => True (1 < 2 and 2 < 3)

(True + 1) * 5
# => 10 (please, please, please don't do this)
```

Strings and Lists



```
course = "hap.py code"
```



```
course[start:stop:step]
```

```
course[2]         # => 'p'
course[:4]        # => 'hap.'
course[5:]        # => 'y code'
course[-2]        # => 'd'
course[1:8:2]     # => 'a.yc'
course[8:1:-2]    # => 'o pp'
course[::-1]      # => 'edoc yp.pah'
```

Console I/O

input prompts for input from the terminal

```
>>> name = input("What is your name? ")
What is your name? Unicorn

>>> print("Nice to meet you, ", name)
Nice to meet you, Unicorn
```

Control Flow

The statement is evaluated as a boolean…

```
if time_in_oven == required_time :
    print("Take it out of the oven!")
elif time_in_oven < required_time:
    print("It's not done yet!")
else:
    print("Uhhh... Hate to break it to you...")
```

If it’s `False`, Python checks `elif` statements sequentially

`elif = else + if`

Otherwise, Python executes the else statement

Control Flow, Addendum

When code doesn’t work at runtime, it’ll raise an `Exception`.
When syntax is incorrect, it’ll raise a `SyntaxError`.

```
n = int(input('How many unicorns would you like? '))
How many unicorns would you like? A ton!
# Raises ValueError (a type of Exception)
```

Solution!

```
while True:
    try:
        n = int(input('How many unicorns would you like? '))
        break
    except ValueError:
        print("Invalid input. Try again...")
```

Control Flow, Addendum

```
try:
    some_dangerous_code()
except SomeError as e:
    handle_the_exception(e)
except AnotherError:
    handle_without_binding()
except (OneError, TwoError):
    handle_multiple_errors()
except:
    handle_wildcard()
```

Bind a name to the exception

Catch multiple exceptions

Wildcard catches everything

Good Python:
Don’t be a Pokémon Trainer!

Control Flow, Addendum

```
while True:
    try:
        n = int(input('How many unicorns would you like? '))
        break
    except:
        print("Invalid input. Try again...")
```

"I’ll just catch ‘em all!”

Uh oh! We can’t use Control-C to exit!

Control Flow, Addendum

A bit of Python philosophy: EAFP is better than LBYL.

It's **easier to ask forgiveness than for permission** is better than **look before you leap**.

Translation: Errors are really lightweight and easy to raise! Use them to handle control flow.

Just open a file instead of checking first that it exists!

Just pop an element; don't check that the list is nonempty.

Raise an error with `raise SomeError`

Helps prevent race conditions but can often be a source of bugs if you forget to handle all potential exceptions.

```
try:
    os.remove(filename)
except FileNotFoundError:
    pass
```

Try to remove `filename`. If that fails, deal with the error

Loops

While loops are very similar to other languages:

```
while condition:
    do_action()
```

For loops are over some collection of items…

```
for item in collection:
    do_action_on(item)
```

…which can be a range object, producing C++/Java-like loops.

```
for i in range(start, stop, step):
    use_number(i)
```

Functions

The def keyword defines functions

Parameters are untyped

```
def is_prime(n):
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
```

All functions return something, even if it’s None

Assignment Expressions

Problem: We want to store an object and use it (maybe in a loop), at the same time.

We want to prompt the user until they enter “Yes” or “No” (in a loop) and also want to keep track of that response.

Must be surrounded by parantheses

```
while (answer := input("Yes/No? ")) not in ['Yes', 'No']:
    print("Please enter either 'Yes' or 'No'.")

answer # => Whatever the valid answer was!
```

Assignment Expressions

First Python evaluates the expression…

```python
while (answer := input("Yes/No? ")) not in ['Yes', 'No']:
    print("Please enter either 'Yes' or 'No'.")

answer # => Whatever the valid answer was!
```

```
Yes/No? I hate yes or no questions...
```

Assignment Expressions

First Python evaluates the expression…

Then Python binds the result to `answer`

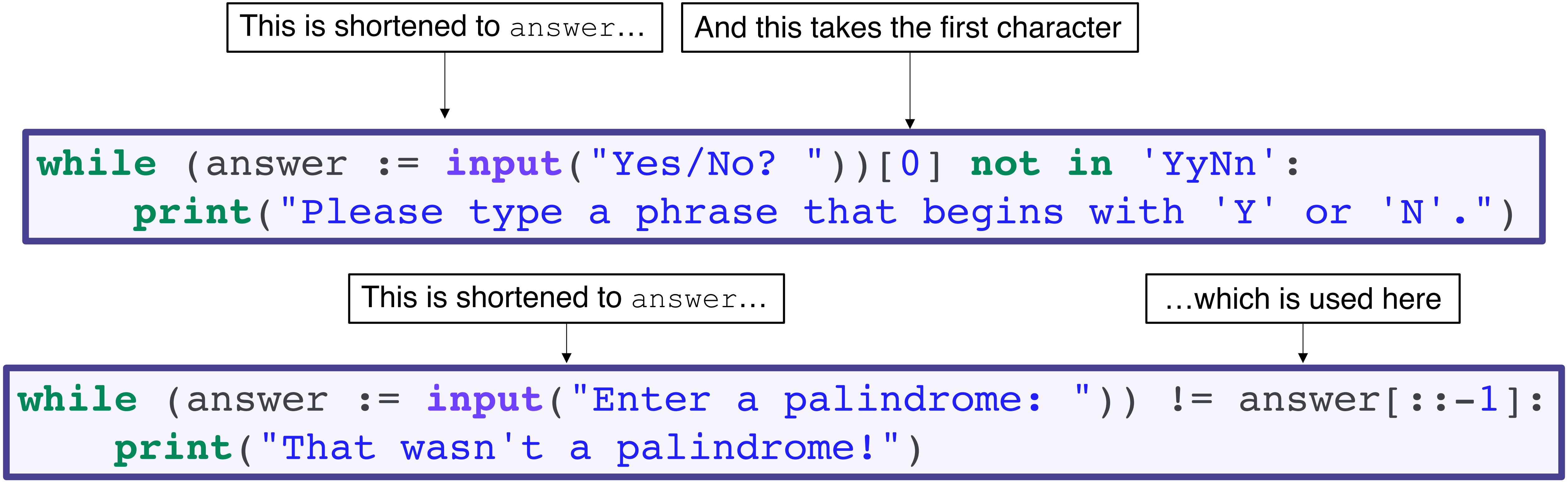
Then, “replaces” the parentheses with `answer`

```
while answer not in ['Yes', 'No']:
    print("Please enter either 'Yes' or 'No'.")

answer # => Whatever the valid answer was!
```

```
Yes/No? I hate yes or no questions...
Please enter either 'Yes' or 'No’.
```

Assignment Expressions

Because of the execution order, you can do operations on the assignment variable without storing them!



This is shortened to answer…

And this takes the first character

```python
while (answer := input("Yes/No? "))[0] not in 'YyNn':
    print("Please type a phrase that begins with 'Y' or 'N'.")
```

This is shortened to answer…

…which is used here

```python
while (answer := input("Enter a palindrome: ")) != answer[::-1]:
    print("That wasn't a palindrome!")
```

Time for new stuff!

More on crazy cool Python basics!

The Data Model

Objects

Everything is an object!

```
isinstance(4, object)                  # => True
isinstance("Michael", object)          # => True
isinstance([4, 5, 'seconds'], object)  # => True
isinstance(None, object)               # => True
isinstance(str, object)                # => True
isinstance(object, object)             # => True
```

Objects have identity, type, and value

Variables are un-typed (dynamically typed)

Objects have identity

When objects are created, they’re given an identity, which never changes.

- In CPython (an implementation of Python), the identity of an object is the *actual* memory address of the object.

The id function returns the object’s “identity.”

```
id(41) # => 4421836688 (e.g.)
```

Objects have type

The type determines what can be done to an object (e.g., does it have a length?)

```
type("unicorn")  # => str
type(1)          # => int
type(3.0)        # => float
```

Types are also objects!

```
isinstance(type('unicorn'), object) # => True
```

Objects have value

Objects contain pointers to their underlying data blob.

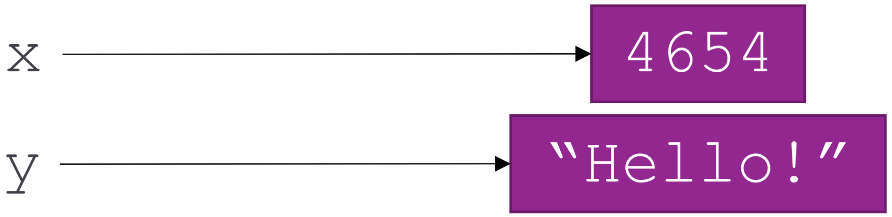
This overhead means that even small things take up a lot of space!

```
(41).__sizeof__() # => 28 (bytes)
```

Variables

Variables are references to objects (little more than a pointer).

```
x = 4654
y = "Hello!"
```



x

y

4654

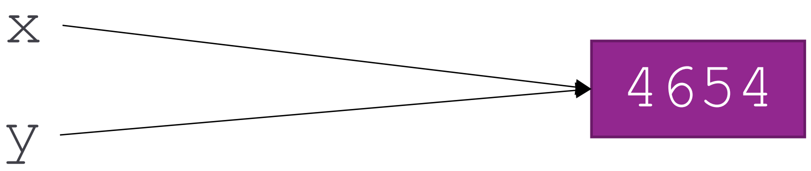
“Hello!”

Variable assignment does **not** copy the object.

It adds another reference to the same object.

Python will **always** handle the creation of new objects.

```
x = 4654
y = x
```



x

y

4654

Variables

Remember “Namespaces are one honking great idea!”?

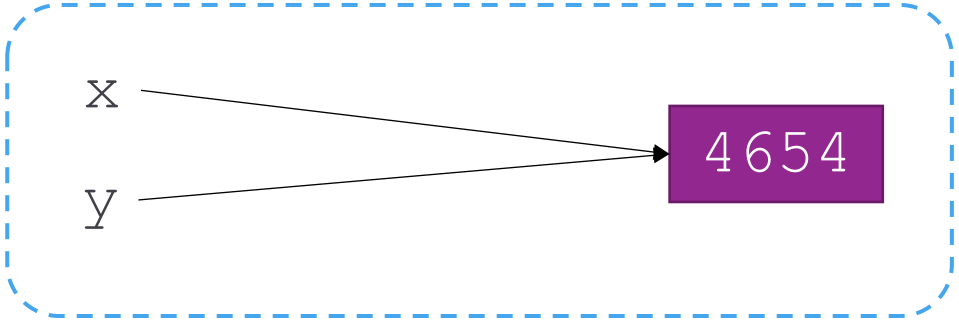
A Python namespace maintains information about variables and their associations. (Kind of like “scope” in other languages)

The namespace is implemented using a dict, and there are several: local, global, module, and more!

`locals()`, `globals()`, etc.

We’ll learn more about dicts next week!

```
x = 4654
y = x
```



A namespace tracks associations between variables and objects

Another piece of Python Philosophy:

Duck Typing

When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.

James Whitcomb Riley

Duck Typing

```python
def compute(a, b, c):
    return (a + b) * c

compute(4, 1, 3)        # => 15
compute([1], [2, 3], 2) # => [1, 2, 3, 1, 2, 3]
compute('l', 'olo', 4)  # => 'lolololololololo'
```

Write code which does not look at an object’s type to determine if it has the right interface.

Instead, the method or attribute is simply called or used.

All that matters is that `compute`’s arguments support + and *

Duck Typing

If you can `walk`, `swim`, and `quack`, then you’re a `Duck`

Promotes interface-style generic programming.

We’ll see more later – stay tuned!

Aside: `is` vs. `==`

`is` vs. `==`

We've seen `==` for equality testing

```
1 == 1.0
```

but we know these are different… they're different *objects*

```
type(1) != type(1.0)
int != float
```

The `is` operator checks *identity* instead of *equality.*

```
1 is not 1.0
```

`a is not b`
is syntactic sugar for
`not (a is b)`

When comparing `None` against other singletons, always use `is None` instead of `== None`.

Identity Crisis

```
x = "cs41 rocks!"
y = "cs41 "
y += "rocks!"

x == y # => True
x is y # => False

id(x)  # => 4512586800
id(y)  # => 4512586672

[1, 2, 3] is [1, 2, 3] # => False
```

Almost always

Use `==` when comparing values

Use `is` when comparing identities

Almost never

Strings, Revisited

Special Characters

```
print('doesn\'t') # => doesn't
print("doesn't")  # => doesn't

print('"Yes," he said.')   # => "Yes," he said.
print("\"Yes,\" he said.") # => "Yes," he said.

print('"It isn\'t," she said.') # => "It isn't," she said.
```

Just choose the easiest delimiter to work with!

Useful String Methods

```
greeting = "Hello! Love, unicorn.  "

greeting[4]        # => 'o'
'corn' in greeting # => True
len(greeting)      # => 23

greeting.find('lo')             # => 3 (-1 if not found)
greeting.replace('ello', 'iya') # => Hiya! Love, Unicorn.
greeting.startswith('Hell')     # => True
greeting.endswith(' ')          # => True
greeting.isalpha()              # => False
```

Useful String Methods

```
greeting = "Hello! Love, unicorn.  "

greeting.lower()        # => 'hello! love, unicorn.'
greeting.title()        # => 'Hello! Love, Unicorn.'
greeting.upper()        # => 'HELLO! LOVE, UNICORN.'
greeting.strip()        # => 'Hello! Love, unicorn.'
greeting.strip('.nrH ') # => 'ello! Love, unico'
```

Lists <—> Strings

```
list('Hair toss!')
# => ['H', 'a', 'i', 'r', ' ', 't', 'o', 's', 's', '!']

# `.split` partitions by a delimiter...
'ham cheese bacon'.split()
# => ['ham', 'cheese', 'bacon']

# ...which can be specified, but defaults to whitespace
'3-14-2015'.split(sep='-')
# => ['3', '14', '2015']

# `.join` creates a string from a list of strings
', '.join(['Zheng', 'Antonio', 'Sam'])
# => 'Zheng, Antonio, Sam'
```

String Formatting

```
# Curly braces are placeholders
'{} {}'.format('beautiful', 'unicorn') # => 'beautiful unicorn'

# Provide values by position or placeholder
'{0} can be {1} {0}, even in summer!'.format('snowmen', 'frozen’)
# => 'snowmen can be frozen snowmen, even in summer!'

'{name} loves {food}'.format(name='Michael', food='applesauce')
# => 'Michael loves applesauce' (he does)

# Values are converted to strings
'{} squared is {}'.format(5, 5**2) # => '5 squared is 25'
```

String Formatting

```
# You can use C-style specifiers too!
"{:06.2f}".format(3.14159) # => '003.14'

# Padding can be specified as well.
'{:10}'.format('left') # => 'left      '
'{:*^12}'.format('CS41') # => '****CS41****'

# You can even look up values!
captains = ['Kirk', 'Picard']
"{caps[0]} > {caps[1]}".format(caps=captains)
```

(Other Options for) String Formatting

```python
# String concatenation with +
"I am " + str(age) + " years old."

# Formatted string literals (only on Python 3.6+)
f"I am {age} years old."
f"{', '.join(['Zheng', 'Antonio', 'Sam'])} are awesome!"
```

`.format` is generally the safest, fastest option

Break for “Half”time!

Announcements

Piazza	Sign up!
Auditors	Email us so we can add you to our internal lists.
Axess	Enrollment codes!
Materials	Slides always, videos with best effort :)
Assignment 0	Warm up, check installation & submission (link).
Python 3.8	Set up Python before or during Lab 2 (Week 2)

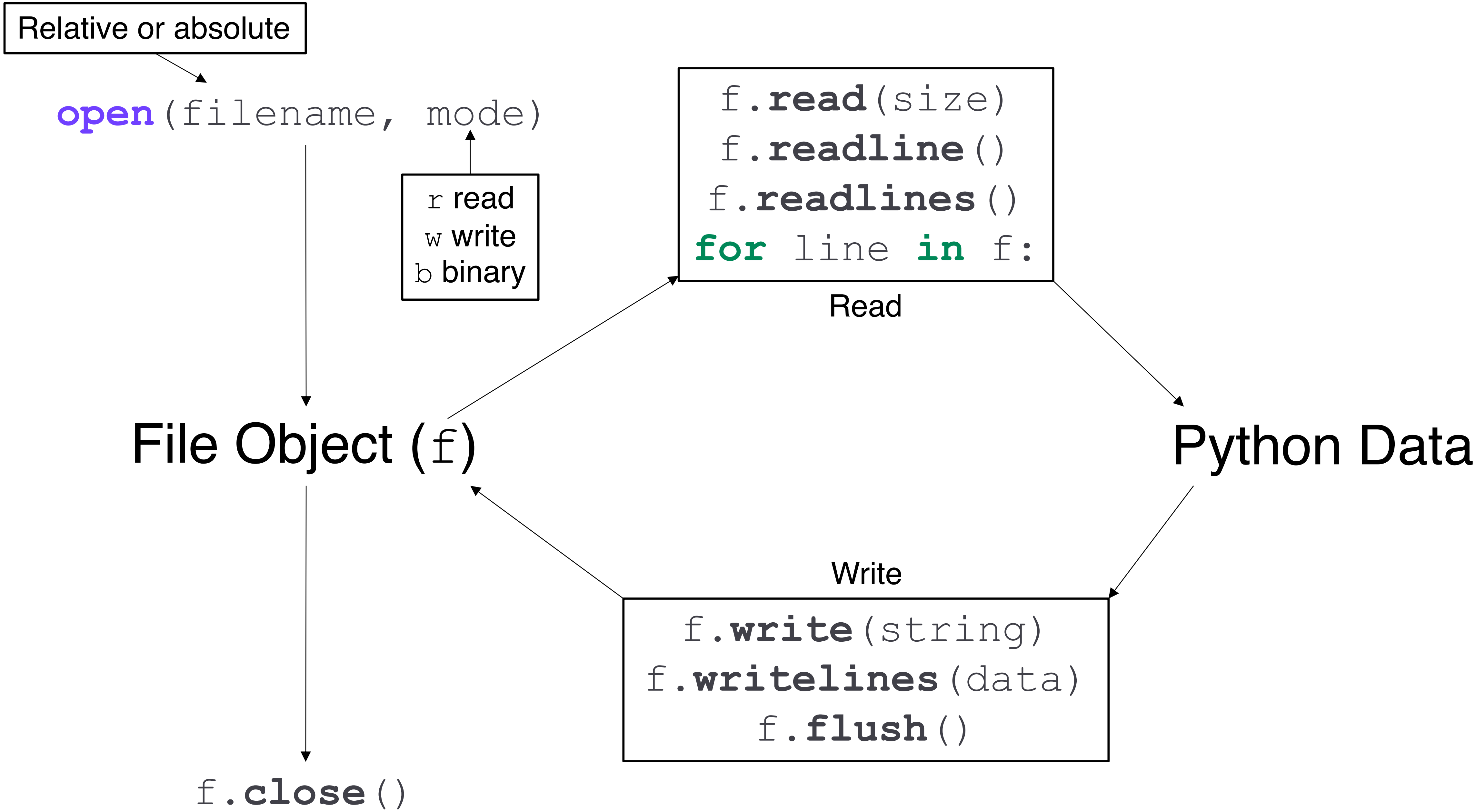
Onwards and Upwards!

Week 1 of CS 41

Today in CS 41

- Brief Review
- The Data Model
- String Formatting
- File I/O
- Modules
- Virtual Environments

File I/O



Relative or absolute

```
open(filename, mode)
```

`r` read
`w` write
`b` binary

File Object (`f`)

Read

```
f.read(size)
f.readline()
f.readlines()
for line in f:
```

Python Data

Write

```
f.write(string)
f.writelines(data)
f.flush()
```

```
f.close()
```

A Motivating Example

```python
f = open('knights.txt')
for line in f:
    data = line.split()

    name = data[0]
    wins = int(data[1])
    losses = int(data[2])

    win_percent = 100 * wins / (wins + losses)
    print(f"{name}: Wins {win_percent:.2f}%")
f.close()
```

```
Lancelot 6 0
Galahad 7 12
Geraint 3 1
Mordred 0 0
```

knights.txt

Something goes wrong…

A Motivating Example

```
f = open("file.txt", "r")
print(1 / 0)  # Crash!
f.close()  # Never executes!
```

We never close the file! That’s bad!

```
with open('file.txt', 'r') as f:
    content = f.read()
    print(1 / 0)
```

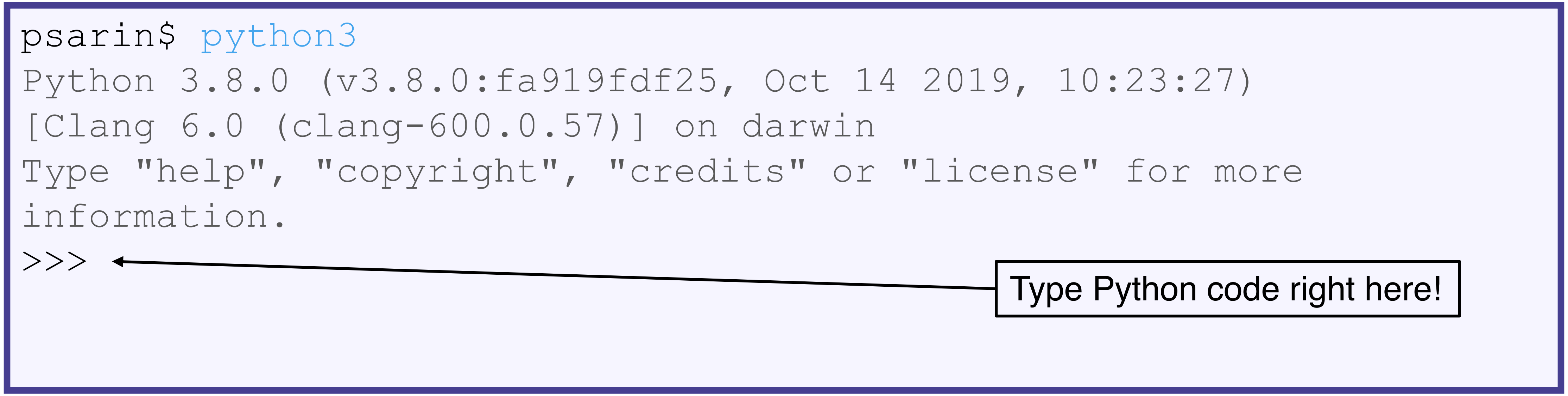
`with expr as var` ensures that `expr` will be “entered” and “exited” regardless of the code block execution

Be responsible:
Use context management to prevent sad unicorns!

```
with open('file.txt', 'r') as f:
```

Modules

So far: The Interactive Interpreter



```
psarin$ python3
Python 3.8.0 (v3.8.0:fa919fdf25, Oct 14 2019, 10:23:27)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

Type Python code right here!

Problem: Code is temporary!
Solution: Write the code in a file!

In Files

hello.py

```python
#!/usr/bin/env python3
"""Ask the user's name and greet them."""

def greet(name):
    print("Hey, {}! I'm Python.".format(name))

def main():
    name = input("What is your name? ")
    greet(name)

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

Shebang specifies executables and options

`__name__` is set to `'__main__'` if the file is executed as a script

Running Scripts

```
psarin$ python3 my_script.py
<output from the script>

psarin$ python3 hello.py
What is your name? Unicorn
Hey Unicorn! I’m Python.

psarin$
```

Running Scripts: Interactive Mode

```
psarin$ python3 -i hello.py
What is your name? Unicorn
Hey Unicorn! I’m Python.
>>> greet('Michael')
Hey Michael! I’m Python.
>>>
```

Super useful for debugging!

We’ll see more ways to debug… Stay tuned!

Running Scripts as Executables

```
psarin$ chmod +x hello.py
psarin$ ./hello.py
What is your name? Unicorn
Hey Unicorn! I’m Python.

psarin$
```

The shebang line specifies how the script should be run, when it’s called as an executable

Using Modules

```python
# Import a module.
import math
math.sqrt(16) # => 4.0

# Import specific symbols from a module (though we usually import
# the entire module).
from math import ceil, floor
ceil(3.7)  # => 4.0
floor(3.7) # => 3.0

# Bind module symbols to a new symbol in the local namespace.
from some_module import super_long_symbol_name as short_symbol
import why_did_anyone_name_a_module_this_long as short_module

# *Any* python file (including those you write) is a module.
from my_file import my_fn, my_variable
```

Virtual Environments

What is a virtual environment?

A local, isolated Python environment.

- Can run an isolated interpreter environment…
- …install third party libraries…
- …and write/run scripts.

But… why?

- Imagine one application uses SuperCoolLibrary v1 but another uses SuperCoolLibrary v2.
- We’ll use Python 3, but many computers default to Python 2.7.
- Solution: Create an isolated sandbox for this course.

An Analogy: Building a Unicorn Shelter

Unicorn World

Default Toolshed

Rotten Wood
Un-magical Shingles (the magic wore off)
Broken Hammer
Rusted Nails

My Unicorn Shelter

Wood? Rotten
Nails? Rusted
Shingles? Not magical!

We want to build a unicorn shelter, but we don’t want to use the default tools!

An Analogy: Building a Unicorn Shelter

Unicorn World

My Unicorn Shelter

New Wood
Magical Shingles
Good Hammer
Shiny Nails

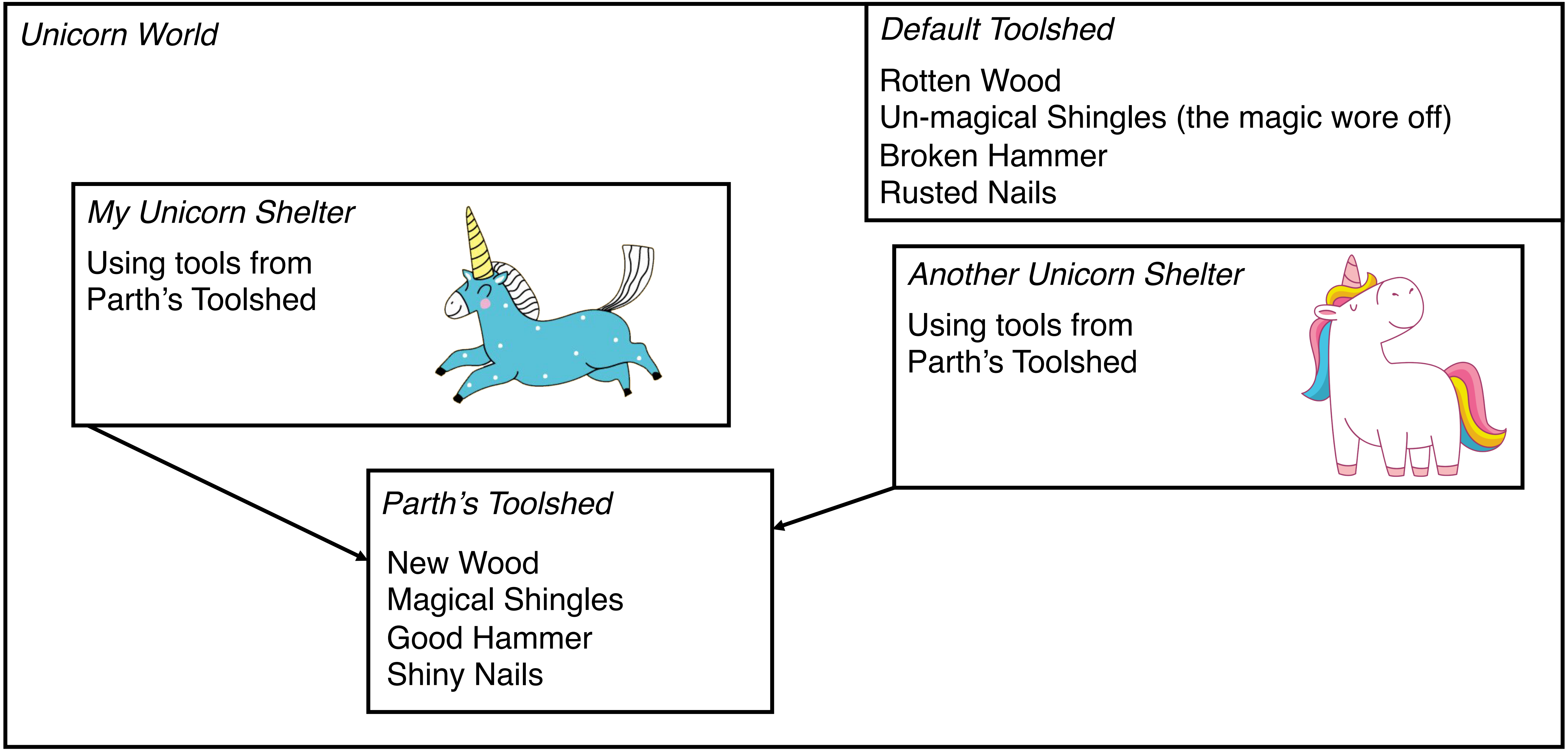
Default Toolshed

Rotten Wood
Un-magical Shingles (the magic wore off)
Broken Hammer
Rusted Nails

But, what if we want to build a new unicorn shelter? We need some way to **share the new tools**

Solution 1: Get new tools and keep them in my shelter, **where I'm working**

An Analogy: Building a Unicorn Shelter



Unicorn World

Default Toolshed

Rotten Wood
Un-magical Shingles (the magic wore off)
Broken Hammer
Rusted Nails

My Unicorn Shelter

Using tools from Parth’s Toolshed

Another Unicorn Shelter

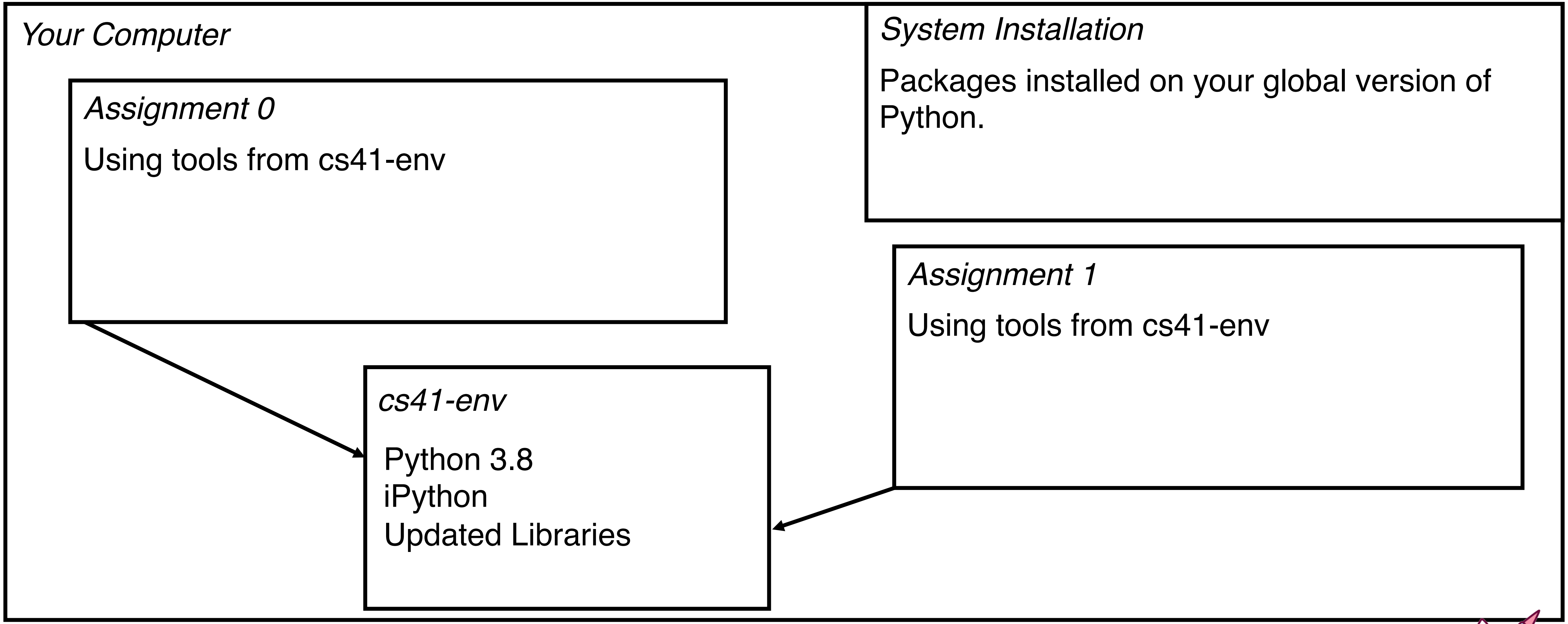
Using tools from Parth’s Toolshed

Parth’s Toolshed

New Wood
Magical Shingles
Good Hammer
Shiny Nails

Solution 2: Put the tools in a **toolshed**

Virtual Environments: Unicorn Shelters in Practice



Your Computer

Assignment 0

Using tools from cs41-env

System Installation

Packages installed on your global version of Python.

Assignment 1

Using tools from cs41-env

cs41-env

Python 3.8
iPython
Updated Libraries

Unicorn, watching you code

How do I get new tools?

Use `pip`! It’s the preferred package manager.

```
pip install numpy
```

When you can, use pip instead of:

`conda` – less flexible, less supported

`pipenv` — newer, less stable

`python setup.py install` — building from source (longer, riskier)

High Level: Setting up the Toolshed

1. Install Python 3.8
2. Create a *virtual environment* that uses Python 3.8

   (and learn how to activate/deactivate the virtual environment)
3. Install and upgrade packages in the virtual environment

Optional: Use `virtualenvwrapper` for managed environments.

Detailed instructions online!

Next Time

Transition

Moving from Python *basics and syntax* to *tools and tricks*.

Week 2: Data Structures

Week 3: Functions

Week 4: Functional Programming

Week 5: Python & the Web