GENERAL ASSEMBLY

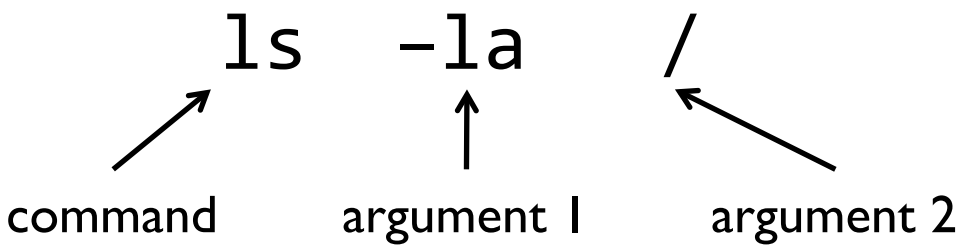
CMDLINE REVIEW

REVIEW

- How Terminal Works
- File Encodings

HOW THE TERMINAL WORKS

- When a user enters a command, the shell successively searches each directory in in $PATH until it finds an executable of the same name.
- Then, it runs that executable with the given arguments.



HOW THE TERMINAL WORKS

- When a user enters a command, the shell successively searches each directory in in $PATH until it finds an executable of the same name.
- Then, it runs that executable with the given arguments.

```
>> echo $PATH
>> /Users/danwilhelm/anaconda/bin:/Library/Frameworks/Python.framework/Versions/3.4/bin:/Users/danwilhelm/.rbenv/shims:/Users/danwilhelm/.rbenv/shims:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/opt/X11/bin:/usr/local/git/bin:/Users/danwilhelm/.rvm/bin
```

HOW THE TERMINAL WORKS

- When the terminal application is opened, several scripts are run, including ~/.bash_profile (if using the Bash shell).

- This is how directories are added to $PATH, colors are set up, etc.

HOW THE TERMINAL WORKS

- When the terminal application is opened, several scripts are run, including ~/.bash_profile (if using the Bash shell):

```
export PS1="\[\033[36m\]\u\[\033[m\]@\[\033[32m\]\h:\[\033[33;1m\]\w\[\033[m\]\$ „
export CLICOLOR=1
export LSCOLORS=ExFxBxDxCxegedabagacad

alias ls="ls -G"

# added by Anaconda3 2.2.0 installer (prepends anaconda to $PATH)
export PATH="/Users/danwilhelm/anaconda/bin:$PATH"
```

- This is how directories are added to $PATH, colors are set up, etc.

ENCODINGS

- All files are just sequences of numbers. So, all files look alike.
- Hence, the operating system needs a way to know how to interpret a file's contents.

ENCODINGS

- All files are just sequences of numbers. So, all files look alike.
- Hence, the operating system needs a way to know how to interpret a file's contents.
- There are only two main ways to do this:
  1. The file extension provides a hint at how to interpret the file, e.g.:

     .jpeg .htaccess .txt .docx
  2. The beginning of the file could provide clues, e.g.:

     all JPEG files start with the sequence: 255, 216

ASCII VS UNICODE

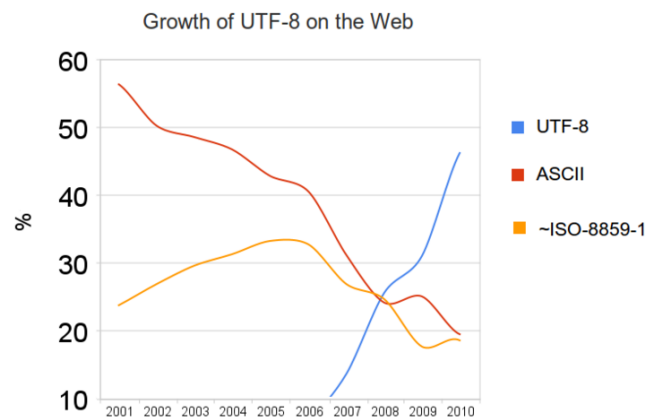
- Text files (.txt, .py, .html) have historically used the 7-bit **ASCII** encoding
- However, ASCII only supports the English alphabet. So, many other text encodings were created.
- Collectively, these alternative encodings are known as **Unicode**.

ASCII VS UNICODE

- Text files (.txt, .py, .html) have historically used the 7-bit **ASCII** encoding
- However, ASCII only supports the English alphabet. So, many other text encodings were created.
- Collectively, these alternative encodings are known as **Unicode**.
- *Problem*: Nowadays, text files could be encoded via ASCII or one of many Unicode encodings.
- However, there is no way to infer how to interpret the numbers -- the filename extensions are all the same, and text files immediately begin with the text content!

ASCII VS UNICODE

- For example, the most popular encoding is UTF-8. It relies on ASCII being 7 bits and uses the 8th bit as a flag to indicate a special character.
- Hence, it is backwards compatible with ASCII. An ASCII file is also a UTF-8 file! (Assuming the 8th bits are all zero.)



ASCII VS UNICODE

- For example, the most popular encoding is UTF-8. It relies on ASCII being 7 bits and uses the 8th bit as a flag to indicate a special character.
- Hence, it is backwards compatible with ASCII. An ASCII file is also a UTF-8 file! (Assuming the 8th bits are all zero.)
- If an ASCII file has a non-zero 8th bit, this will throw a Python exception saying it is invalid ASCII!
- If a UTF-8 encoded-file does not follow the UTF-8 rules, then it will also throw a Python exception, e.g. if Byte 1 is 110xxxxx, the next byte *must be* 10xxxxxx, where ‘x’ is any bit.

ASCII VS UNICODE

- If we ask Python to interpret a file using a certain encoding and the file does not follow these rules, then an exception will be thrown.
- This is unfortunately the only way to know whether we guess the encoding correctly – attempt to interpret the file numbers in one way and see if it follows the encoding's rules.

ASCII VS UNICODE

- If we ask Python to interpret a file using a certain encoding and the file does not follow these rules, then an exception will be thrown.
- This is unfortunately the only way to know whether we guess the encoding correctly – attempt to interpret the file numbers in one way and see if it follows the encoding's rules.
- So, it is preferred to try opening the file using different encoding. If this does not work, you can choose to ignore errors.
- Note: When you open a file in an IDE or text editor and save it, the IDE may resave the file in a different encoding!

UPDATING CONDA PACKAGES

```
conda update --all
```

GENERAL ASSEMBLY

INTRO TO PYTHON

INTRO TO PYTHON

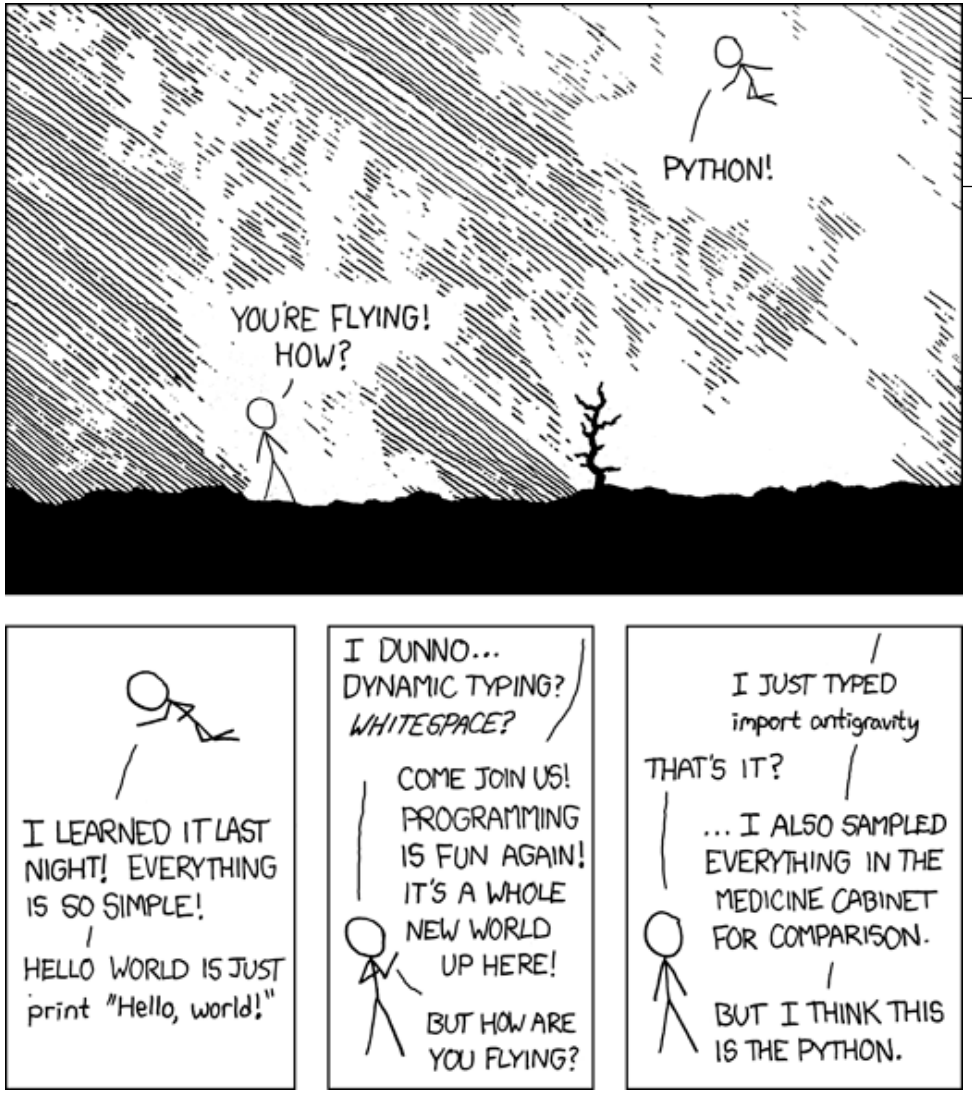
- What is Python?
- Python Fundamentals

PEP 20 (THE ZEN OF PYTHON)

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Readability counts.

https://www.python.org/dev/peps/pep-0020/

Also see PEP 8 (Style Guide for Python Code)
https://www.python.org/dev/peps/pep-0008/



https://xkcd.com/353/

PYTHON LANGUAGE FEATURES

- Interpreted
- High-level
- Emphasizes readability

- Supports many programming paradigms – e.g. object-oriented, imperative, functional, and procedural.

- Dynamic typing
- Strongly typed
- Automatic memory management

WHY DO PEOPLE USE PYTHON?

- Open source (not locked in to one company)
- Clean syntax emphasizes readability
  - Great for scripting (interpreted & tolerant to errors)
- Modules and Packages
  - Allows easier code reuse
  - Large community – has many prewritten modules!
- Increased productivity
  - No compilation step, so edit-test-debug is fast
  - Powerful debugging capabilities (no segmentation faults)

WHEN SHOULD YOU NOT USE PYTHON?

- Performance matters.
- Low memory usage is important.
- Direct access to hardware required.
- Want OS-specific GUI features.
- Programming embedded systems.

PYTHON 2 VS PYTHON 3

> "Short version: Python 2.x is legacy, Python 3.x is the present and future of the language" - https://wiki.python.org/moin/Python2orPython3

Are you new to Python?

Learn version 3.

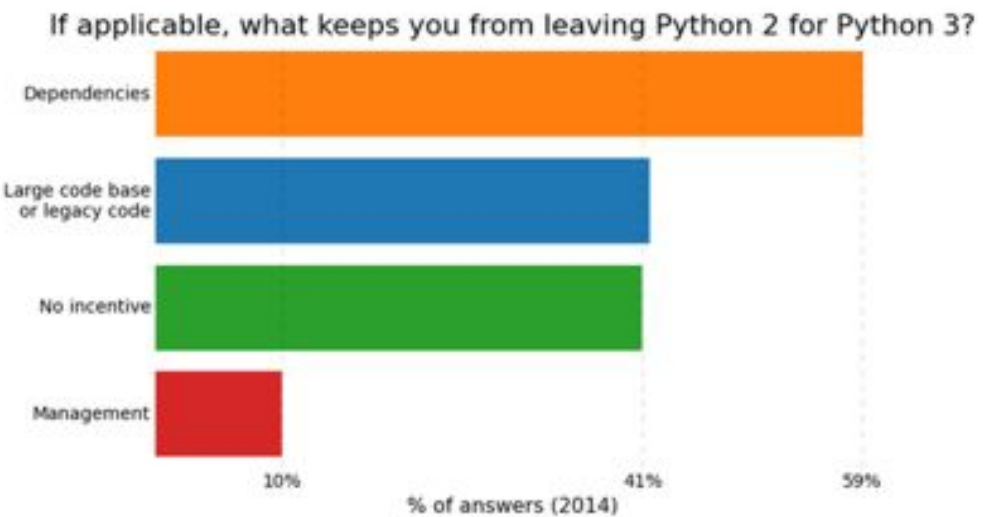
Are you writing a new program?

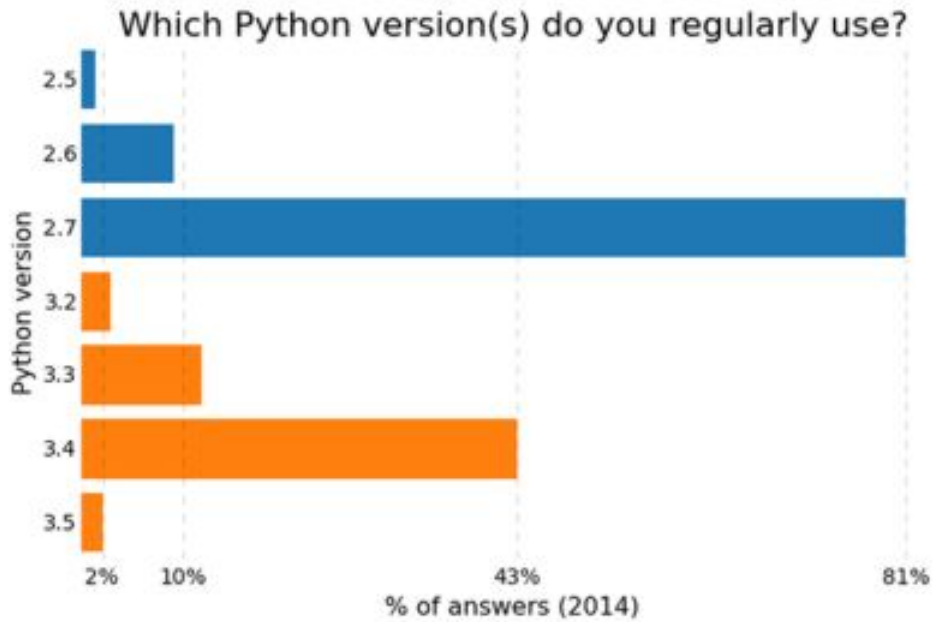
Research whether the libraries you want to use support Python 3. (They probably do.) If so, use Python 3!

Are you maintaining an old project?

Use the version of Python they use.

PYTHON 2 VS PYTHON 3 (FEB 16, 2015)





http://www.i-programmer.info/news/98-languages/8269-python-2-versus-python-3-revisited.html

LEARNING PYTHON

- Official Tutorial
  https://docs.python.org/3/tutorial/index.html
- Google's Python Class
  https://developers.google.com/edu/python/
- Learn Python the Hard Way
  http://learnpythonthehardway.org/book/
- Most importantly, MAKE THINGS!

Insert class title

PYTHON FUNDAMENTALS

PIP INSTALLS PACKAGES

- **pip** – A package management system used to install and manage software packages written in Python.
- If a package is available on Anaconda, install via conda.
- If a package is not available on Anaconda:

```
pip install money
pip uninstall money
```

DATA TYPES

- Booleans (True or False)
- Numbers
- Strings
- Lists
- None
- Tuples
- Sets
- Dictionaries

NEXT WEEK:

- ndarray (from numpy)
- Series (from pandas)
- DataFrame (from pandas)

PYTHON IDENTIFIERS

- Python is not as expansive as you may think.
- Every identifier you see (without any imports) is either an operator, reserved word, or built-in! So, there are a limited number of things to know.

PYTHON IDENTIFIERS

- Python is not as expansive as you may think.
- Every identifier you see (without any imports) is either an operator, reserved word, or built-in! So, there are a limited number of things to know.

- An identifier is:

A single letter or underscore, followed by zero or more letters, numbers, or underscores.

```
(letter | _) (letter | number | _)*
```

RESERVED WORDS/KEYWORDS

- Reserved words require special syntax to be used around the word – their syntax is unique compared to the rest of Python.
- Reserved words cannot be used as variables.

```
False      class      finally    is         return
None       continue   for        lambda     try
True       def        from       nonlocal   while
and        del        global     not        with
as         elif       if         or         yield
assert     else       import     pass
break      except     in         raise
```

https://docs.python.org/3/reference/lexical_analysis.html#keywords

BUILT-IN FUNCTIONS

		Built-in Functions		
abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

https://docs.python.org/3/library/functions.html

STATEMENTS VS EXPRESSIONS

An **expression** is valid code that evaluates to a value:

```
>>> 10 + 7
>>> range(10)
```

A **statement** is a line of code that performs an action, e.g.:

```
>>> print(10 + 7)
>>> x = 10 + 7
```

Programs are collections of statements.

CONDITIONALS

```
>>> donuts = 5
>>> if donuts < 2:
...     print(“Good job!”)
... else:
...     print(“You ate too many donuts.”)
```

- According to PEP 8, use 4 spaces per indentation level.
- Use ‘=‘ for assignment and ‘==‘ for comparison.
- Name your variables well! Why is “donuts” a poor name?

STRINGS

```
>>> x = input(“Exit program? (yes/no)”)
>>> if x.lower() == “yes”:
...     print(“Exiting the program ...”)
```

- Notice the method call. What does it do? Why is it done?
- Use ‘=‘ for assignment and ‘==‘ for comparison.
- Is x a good variable name? What should it be?
- ‘ vs “ vs “””
- Indexing characters.

LISTS VS TUPLES

- A list contains ordered data, typically of the same data type:

```
>>> x = [“Tim”, “Sandy”, “Martin”, “Shawna”]
>>> print(x[0])
```

- A tuple contains ordered groups of variables, often of different data types:

```
>>> x = (“Tim”, 5)   # Note (name, age) describe one person
>>> print(x)
```

LOOPING

- Given a list or tuple, we often want to do something with each member. To do this, we loop through the list:

```
>>> names = [“Tim”, “Sandy”, “Martin”, “Shawna”]
>>> for name in names:
...     print name
```

- The indented code in the for block is run for each item in the list. Each time the indented code is run, *name* refers to the next item in the list. Note that *name* can be any name.

FUNCTIONS

- A function allows us to take complex code and refer to it in an easy way. For example, "x % 2 == 1" is hard to understand by beginners if encountered in a program. Hence, to make the program easier to read, we refer to what the code does in English, as part of a function call that returns a value:

```
>>> def is_odd(x):
...     return (x % 2 == 1)
```

- What is a better name for x?

HOW TO CODE

When coding, you only have a limited number of options:

- Use a built-in function.
- Use a method or operator of the object.
- Import a package.
- Write your own function.

PRACTICE

CODING CHALLENGE!

INTERACTING WITH THE COMMAND LINE

- Retrieving command-line arguments is easy!

```
import sys
sys.argv    # ARGument Vector (list)
```

Example: python test.py 1 2 3

sys.argv: ['test.py', '1', '2', '3']

INTERACTING WITH THE COMMAND LINE

- Reading from stdin is just like reading a file!

```
import sys
for line in sys.stdin:
    print(line.strip())
```

PYTHON + DATA SCIENCE TRICKS

LISTS VS TUPLES

- Lists: Mutable (can be altered), typically homogenous values
- Tuples: Immutable, typically groups of items that go together

```
>>> people = [('Tim', 32), ('Sandy', 45)]
>>> points = [(0,0,0), (5,4,1), (7,7,8)]
>>> menu_item, price = ('Burger', 2.99)   # unpacking
>>> def max_population(cities):   # multiple return values
        ...
        return (city_name, population)
```

FILES

```
# ‘with’ insures the file is closed if an exception occurs
with open(‘test.txt’, ‘r’) as fin:
    for line in fin:
        print(line)

# Write list of strings ‘lines’ to the file
with open(‘test.txt’, ‘w’) as fout:
    for line in lines:
       fout.write(line + “\n”)
```

QUOTES

- ‘vs “ ← same – both support escape characters (“\n“)
- ‘‘‘ ← triple quotes – allows actual newlines

- \ ← if at the end of a non-quoted line of code, allows you to split the line of code (there is an invisible newline after it)

```
>>> names = “Mike Wallace\nClara Simmons”
>>> names.split(“\n”)
>>> names.replace(“\n”, “, “)
```

STRING FORMATTING

```
“Person #{} is {}”.format(2, ‘George’)

“Person #{num} is {name}”.format(num=5, name=‘Henry’)

“Your price will be ${price:.2f}.”.format(price=4.5127)
```

MODULES

```
import math
>>> math.sqrt(5)

from math import sqrt
>>> sqrt(5)

from math import *
>>> sqrt(5)
>>> tan(5)
```

LIST COMPREHENSIONS

```
cubes = []
for num in range(100):
    cubes.append(num**3)
```

BECOMES

```
cubes = [num**3 for num in range(100)]
```

LIST COMPREHENSIONS

```
cubes = []
for num in range(100):
    if num % 2 == 0:
        cubes.append(num**3)
```

BECOMES

```
cubes = [num**3 for num in range(100) if num % 2 == 0]
```

NOTE you can use these for filtering!

ENUMERATE – WHEN YOU NEED A LOOP INDEX

```
index = 0
for person in people:
        print(“Person #{} is {}”.format(index, person))
        index += 1
```

BECOMES

```
for index,person in enumerate(people):
        print(“Person #{} is {}”.format(index, person))
```

DATES AND TIMES

- `datetime.date -> year, month, day`
- `datetime.time -> hour, minute, second, microsecond, tzinfo (TimeZone INFO)`
- `datetime.datetime -> year, month, day, hour, minute, second, microsecond, tzinfo`

- `datetime.timedelta – difference between dates/times`

EXCEPTIONS

```
try:
        num = int('not an int')
except:   # catches ALL exceptions
        print('Exception caught!')


try:
        num = int('not an int')
except ValueError:   # catches the ValueError exception
        print('Exception caught!')
```

LISTS VS TUPLES

- Lists: Mutable (can be altered), typically homogenous values
- Tuples: Immutable, typically groups of items that go together

```
>>> people = [('Tim', 32), ('Sandy', 45)]
>>> points = [(0,0,0), (5,4,1), (7,7,8)]
```

LISTS VS TUPLES

- Lists: Mutable (can be altered), typically homogenous values
- Tuples: Immutable, typically groups of items that go together

```
>>> people = [('Tim', 32), ('Sandy', 45)]
>>> points = [(0,0,0), (5,4,1), (7,7,8)]
>>> menu_item, price = ('Burger', 2.99)   # unpacking
```

LISTS VS TUPLES

- Lists: Mutable (can be altered), typically homogenous values
- Tuples: Immutable, typically groups of items that go together

```
>>> people = [('Tim', 32), ('Sandy', 45)]
>>> points = [(0,0,0), (5,4,1), (7,7,8)]
>>> menu_item, price = ('Burger', 2.99)   # unpacking
>>> def max_population(cities):   # multiple return values
        ...
        return (city_name, population)
```

FILES

```
# ‘with’ ensures the file is closed if an exception occurs
with open(‘test.txt’, ‘r’) as fin:
    for line in fin:
        print(line)

# Write list of strings ‘lines’ to the file
with open(‘test.txt’, ‘w’) as fout:
    for line in lines:
       fout.write(line + “\n”)
```

OTHER STUFF

- Sets
- Using dictionaries for uniqueness/histograms

EXERCISES!