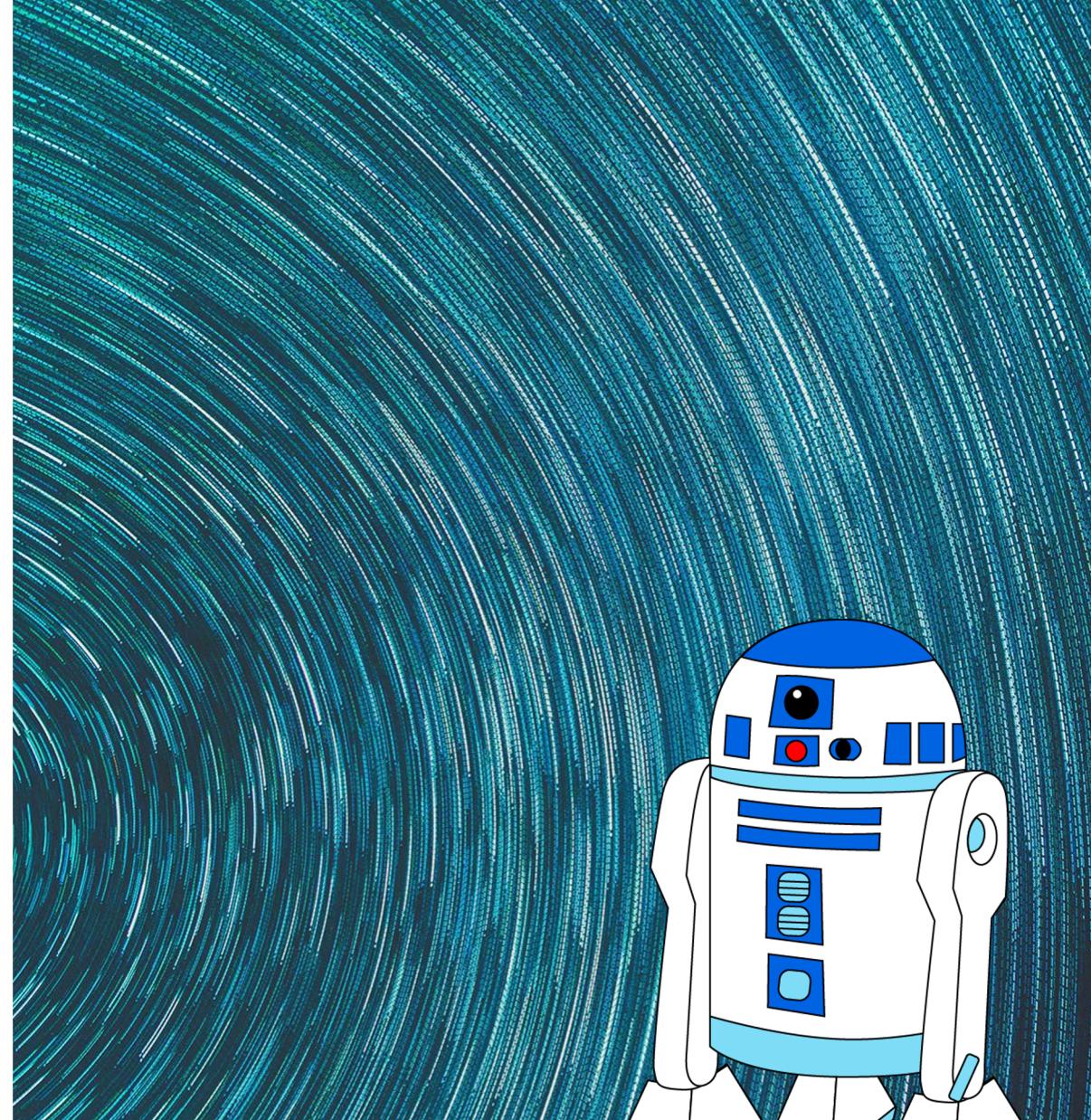


CIS 421/521:
ARTIFICIAL INTELLIGENCE

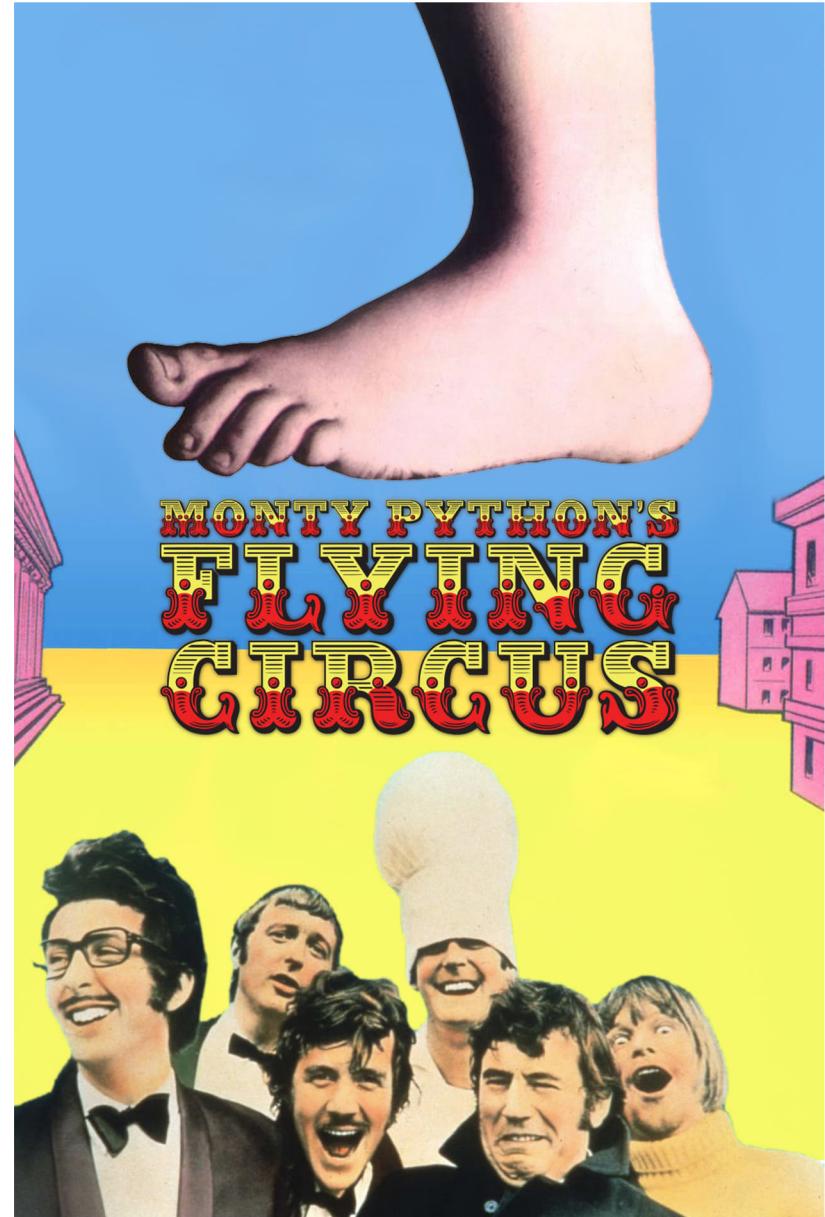
Introduction to Python

Professor Chris Callison-Burch



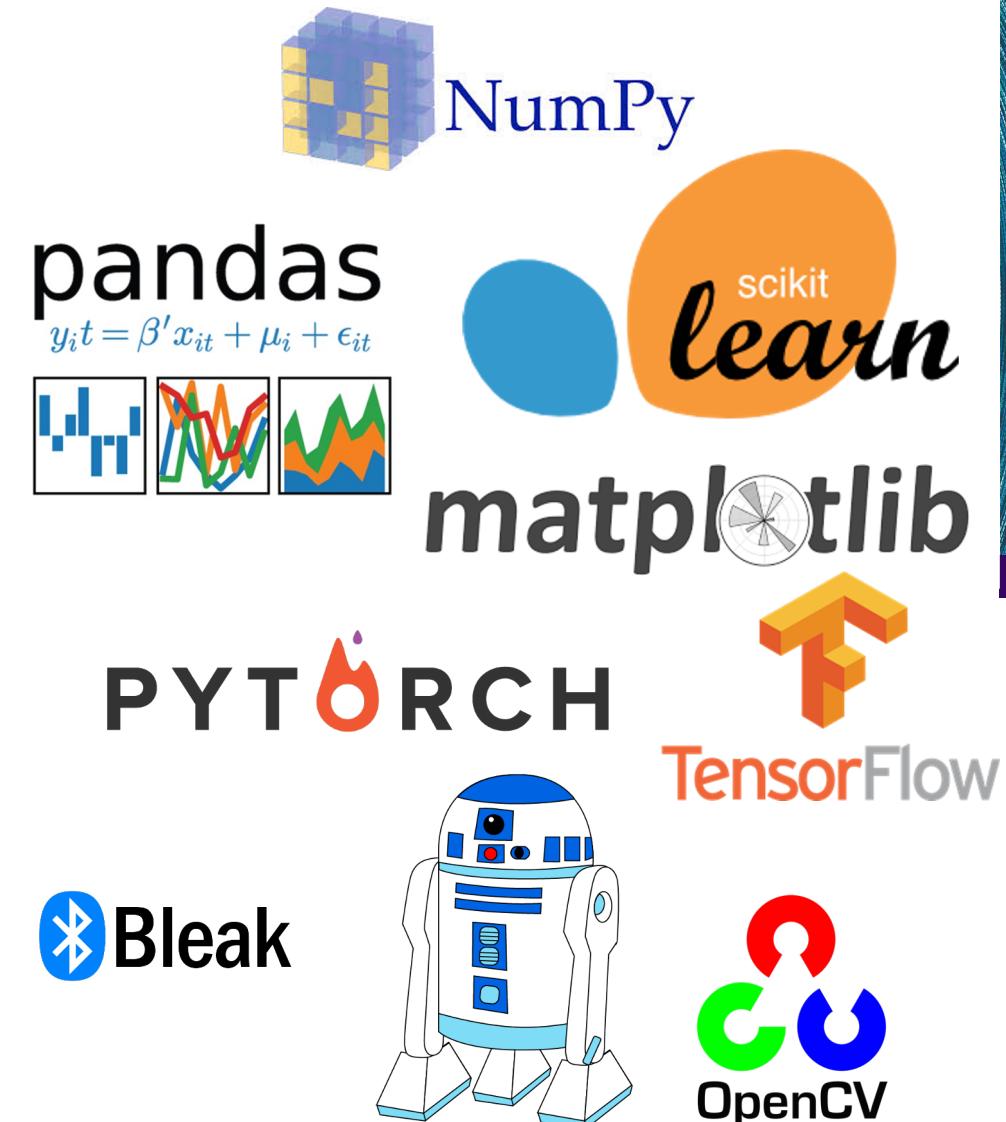
Python

- **Developed by Guido van Rossum in the early 90s**
Originally Dutch, in USA since 1995.
Benevolent Dictator for Life (now retired)
- **Named after the Monty Python comedy group**
- **Download from python.org**



Some Positive Features of Python

- **Fast development:**
Concise, intuitive syntax
 - Whitespace delimitedGarbage collected
- **Portable:**
Programs run on major platforms without change
cpython: common Python implementation in C.
- **Various built-in types:**
lists, dictionaries, sets: useful for AI
- **Large collection of support libraries:**
NumPy for Matlab like programming
Sklearn for machine learning
Pandas for data analysis



Recommended Reading

- **Python Overview**

The Official Python Tutorial (<https://docs.python.org/3/tutorial/index.html>)
Slides for CIS192, Spring 2019
(<https://www.cis.upenn.edu/~cis192/>)

- **PEPs – Python Enhancement Proposals**

[PEP 8](#) - Official Style Guide for Python Code (Guido et al)

- Style is about consistency. 4 space indents, < 80 char lines
- Naming convention for functions and variables: lower_w_under
- Use the automatic pep8 checker!

- PEP 20 – The Zen of Python (Tim Peters) (try: *import this*)

Beautiful is better than ugly; simple is better than complex

There should be one obvious way to do it

That way may not be obvious at first unless you're Dutch

Readability counts

Python REPL Environment

- **REPL**

- Read-Evaluate-Print Loop

- Type “python” at the terminal

- Convenient for testing

- If you’d like syntax highlighting in REPL try [bpython](#)

You should use
Python version 3.7
or version 3.8.

```
cis521x@eniac:~> python3
Python 3.4.6 (default, Mar 22 2017, 12:26:13) [GCC] on linux
Type "help", "copyright", "credits" or license for more information.
>>> print('Hello World!')
Hello World!
>>> 'Hello World!'
'Hello World!'
>>> [2*i for i in range(10)]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
>>> exit()
cis521x@eniac:~>
```

Python Scripts



- **Scripts**

Create a file with your favorite text editor (like Sublime)

Type “python3 script_name.py” at the terminal to run

Not REPL, so you need to explicitly print

Homework submitted as scripts

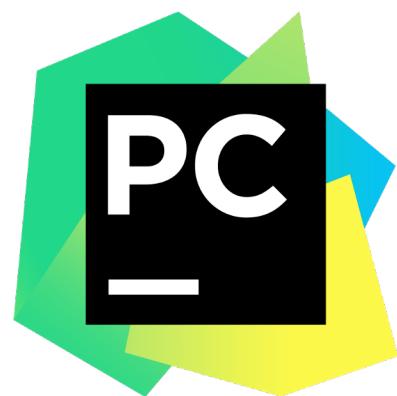
```
cis521x@eniac:~> cat foo.py
import random
def rand_fn():
    """outputs list of 10 random floats between [0.0, 1.0]"""
    return [%.2f % random.random() for i in range(10)]

print ('1/2 = ', 1/2)
if __name__ == '__main__':
    rand_fn()
    print(rand_fn())

cis521x@eniac:~> python3 foo.py
1/2 + 0.5
['0.08', '0.10', '0.84', '0.01', '0.00', '0.59', '0.67', '0.88', '0.58', '0.81']
cis521x@eniac:~>
```

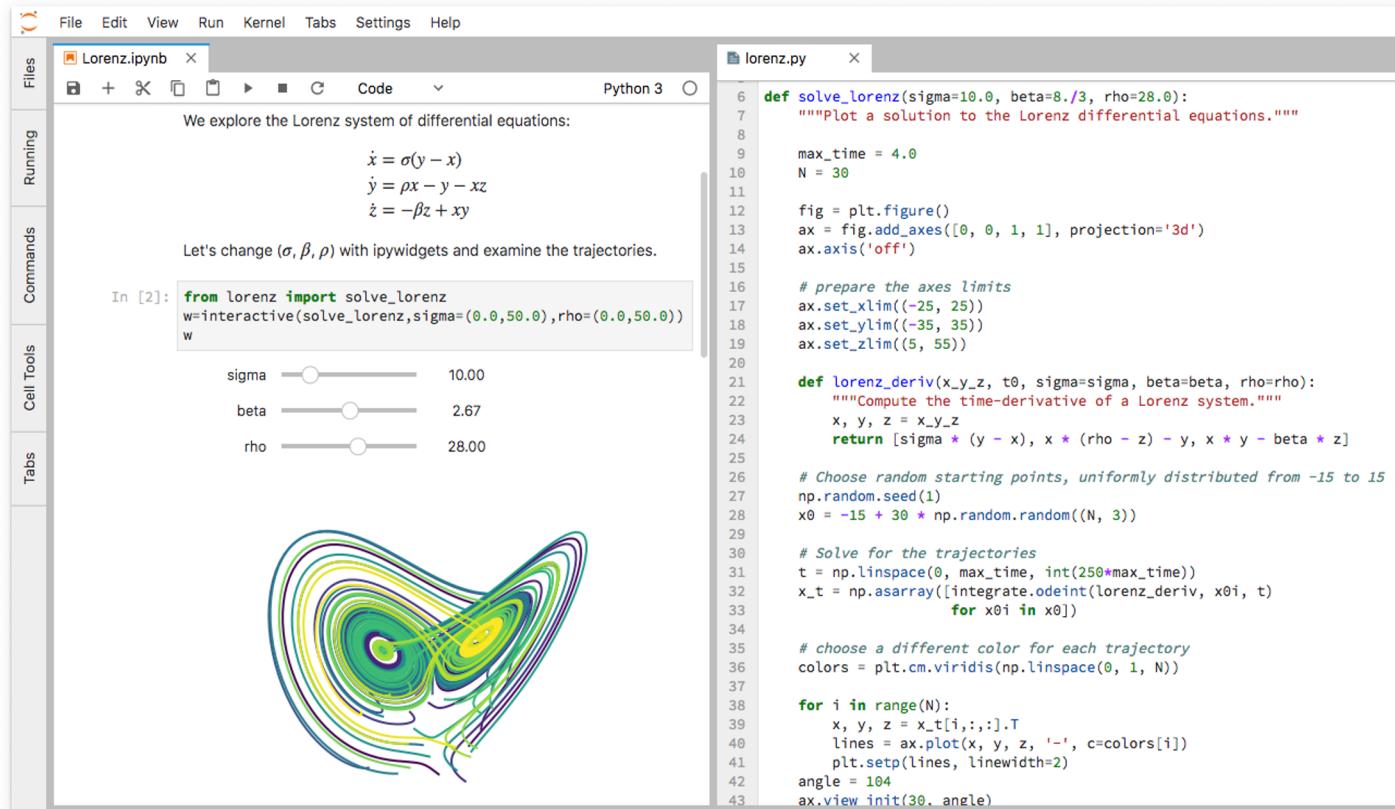
PyCharm IDE

The screenshot displays the PyCharm IDE interface. The top navigation bar shows the project name "djtp_first_steps" and the file "polls/tests.py". The main code editor window contains Python test code for a Django application, specifically for the "polls" app. A search bar at the top of the editor is active, displaying the search term "result". To the right of the editor, a "Database" browser is open, showing the "Django default" database with tables like "auth_group", "auth_user", and "django_admin_log". The bottom of the interface features a "Debug" tool window with tabs for "Debugger", "Console", and "Variables". The "Variables" tab shows local variables for the current test case. The bottom status bar indicates "Tests Failed: 4 passed, 3 failed (4 minutes ago)".



Python Notebooks

- Jupyter Notebooks allow you to interactively run Python code in your web browser and share it with others in places like Google Colab
- They are popular for tutorials since you can include inline text and images



The screenshot shows a Jupyter Notebook interface with two tabs: 'Lorenz.ipynb' and 'lorenz.py'. The 'Lorenz.ipynb' tab contains a cell with text explaining the Lorenz system and three differential equations. It also contains a cell with Python code for an interactive plot and three sliders for parameters sigma, beta, and rho. The 'lorenz.py' tab contains the source code for generating the Lorenz attractor plot.

```
def solve_lorenz(sigma=10.0, beta=8./3, rho=28.0):
    """Plot a solution to the Lorenz differential equations."""

    max_time = 4.0
    N = 30

    fig = plt.figure()
    ax = fig.add_axes([0, 0, 1, 1], projection='3d')
    ax.axis('off')

    # prepare the axes limits
    ax.set_xlim((-25, 25))
    ax.set_ylim((-35, 35))
    ax.set_zlim((5, 55))

    def lorenz_deriv(x_y_z, t0, sigma=sigma, beta=beta, rho=rho):
        """Compute the time-derivative of a Lorenz system."""
        x, y, z = x_y_z
        return [sigma * (y - x), x * (rho - z) - y, x * y - beta * z]

    # Choose random starting points, uniformly distributed from -15 to 15
    np.random.seed(1)
    x0 = -15 + 30 * np.random((N, 3))

    # Solve for the trajectories
    t = np.linspace(0, max_time, int(250*max_time))
    x_t = np.asarray([integrate.odeint(lorenz_deriv, x0i, t)
                     for x0i in x0])

    # choose a different color for each trajectory
    colors = plt.cm.viridis(np.linspace(0, 1, N))

    for i in range(N):
        x, y, z = x_t[i,:,:].T
        lines = ax.plot(x, y, z, 'r', c=colors[i])
        plt.setp(lines, linewidth=2)
        angle = 104
        ax.view_init(30, angle)
```



Structure of Python File

- **Whitespace is meaningful in Python**
- **Use a newline to end a line of code.**
Use \ when must go to next line prematurely.
- **Block structure is indicated by indentation**
The first line with less indentation is outside of the block.
The first line with more indentation starts a nested block.
Often a colon appears at the end of the line of a start of a new block. (E.g. for function and class definitions.)

A Simple Code Sample

```
x = 34 - 23                      # A comment.  
y = 'Hello'                        # Another one.  
z = 3.45  
if x == 3.45 or y == 'Hello':  
    x = x + 1  
    y = y + 'World'                # String concat.  
print(x)  
print(y)
```

```
x = 34 - 23                      # A comment.  
y = 'Hello'                        # Another one.  
z = 3.45  
  
if x == 3.45 or y == 'Hello':  
    x = x + 1  
    y = y + 'World'    # String concat.  
  
print(x)  
print(y)
```

Objects and Types

- **All data treated as objects**
An object is deleted (by garbage collection) once unreachable.
- **Strong Typing**
Every object has a fixed type, interpreter doesn't allow things incompatible with that type (eg. "foo" + 2)
`type(object)`
`isinstance(object, type)`
- **Examples of Types:**
`int, float`
`str, tuple, dict, list`
`bool: True, False`
`None, generator, function`

Static vs Dynamic Typing

- **Java: *static* typing**

Variables can only refer to objects of a declared type
Methods use type signatures to enforce contracts

- **Python: *dynamic* typing**

Variables come into existence when first assigned.

```
>>> x = "foo"  
>>> x = 2
```

`type(var)` automatically determined

If assigned again, `type(var)` is updated

Functions have no type signatures

Drawback: type errors are only caught at runtime

Math Basics

- **Literals**

- **Literals**
 - Integers: 1, 2

- **Literals**
 - FLOATS: 1.0, 2e10

- **Literals**
 - Boolean: True, False

- **Operations**

- **Operations**
 - Arithmetic: + - * /

- **Operations**
 - Power: **

- **Operations**
 - Modulus: %

- **Operations**
 - Comparison: , <=, >=, ==, !=

- **Operations**
 - Logic: (and, or, not) *not symbols*

- **Assignment Operators**

- **Assignment Operators**
 - `+= *= /= &= ...`

- **Assignment Operators**
 - No `++` or `--`

Strings

- **Creation**
 - Can use either single or double quotes
 - Triple quote for multiline string and docstring
- **Concatenating strings**
 - By separating string literals with whitespace
 - Special use of '+'
- **Prefixing with r means raw.**
 - No need to escape special characters: `r'\n'`
- **String formatting**
 - Special use of '%' (as in printf in C)
 - `print("%s can speak %d languages" % ("C3PO", 6000000))`
- **Immutable**

References and Mutability

```
>>> x = 'foo' 
>>> y = x
>>> x = x.strip() # new obj
>>> x
'foo'
>>> y
'foo'
```

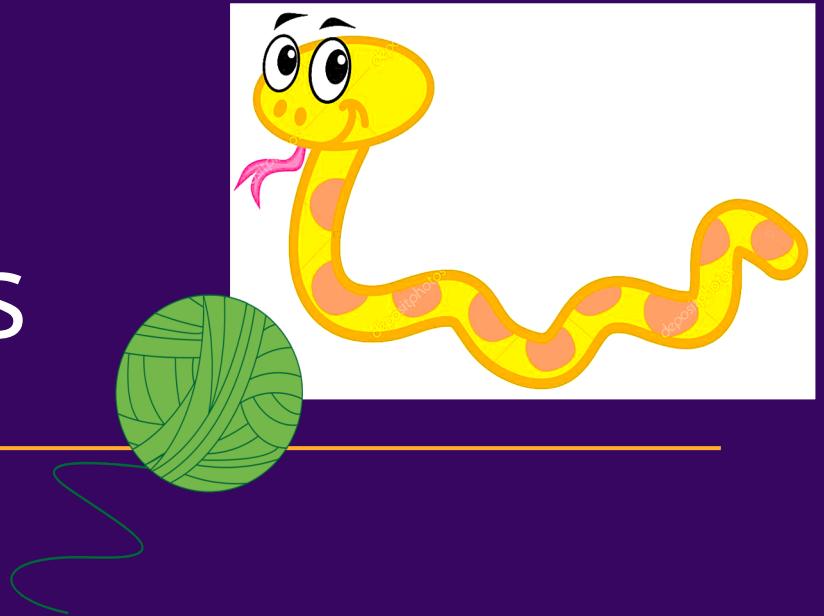
- strings are immutable
- `==` checks whether variables point to objects of the same value
- `is` checks whether variables point to the same object

```
>>> x = [1, 2, 3, 4]
>>> y = x
>>> x.append(5) #same obj
>>> y
[1, 2, 3, 4, 5]
```

- ```
>>> x
[1, 2, 3, 4, 5]
```
- lists are mutable
  - use `y = x[:]` to get a (shallow) copy of any sequence, ie. a new object of the same value

# Sequence Types: Tuples, Lists and Strings

---



# Sequence Types

- **Tuple**

A simple *immutable* ordered sequence of items

*Immutable*: a tuple cannot be modified once created

Items can be of mixed types, including collection types

- **Strings**

*Immutable*

Very much like a tuple with different syntax

Regular strings are Unicode and use 2-byte characters (Regular strings in Python 2 use 8-bit characters)

- **List**

*Mutable* ordered sequence of items of mixed types

# Sequence Types

- The three sequence types share much of the same syntax and functionality.

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def') # tuple
```

```
>>> li = ['abc', 34, 4.34, 23] # list
```

```
>>> st = "Hello World"; st = 'Hello World' # strings
```

```
>>> tu[1] # Accessing second item in the tuple.
```

'abc'

```
>>> tu[-3] #negative lookup from right, from -1
```

4.56

# Slicing: Return Copy of a Subsequence

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

```
>>> t[1:4] #slicing ends before last index
('abc', 4.56, (2,3))
```

```
>>> t[1:-1] #using negative index
('abc', 4.56, (2,3))
```

```
>>> t[1:-1:2] # selection of every nth item.
('abc', (2,3))
```

```
>>> t[:2] # copy from beginning of sequence
(23, 'abc')
```

```
>>> t[2:] # copy to the very end of the sequence
(4.56, (2,3), 'def')
```

# Operations on Lists

```
>>> li = [1, 11, 3, 4, 5]
>>> li.append('a') # Note the method syntax
>>> li
[1, 11, 3, 4, 5, 'a']
>>> li.insert(2, 'i')
>>> li
[1, 11, 'i', 3, 4, 5, 'a']
>>> li = ['a', 'b', 'c', 'b']
>>> li.index('b') # index of first occurrence
1
>>> li.count('b') # number of occurrences
2
>>> li.remove('b') # remove first occurrence
>>> li
['a', 'c', 'b']
```

# Operations on Lists 2

```
>>> li = [5, 2, 6, 8]
>>> li.reverse() # reverse the list *in place* (modify)
>>> li
[8, 6, 2, 5]
```

```
>>> li.sort() # sort the list *in place*
>>> li
[2, 5, 6, 8]
```

```
>>> li.sort(some_function)
sort in place using user-defined comparison
```

```
>>> sorted(li) #return a *copy* sorted
```

# Operations on Strings

```
>>> s = "Pretend this sentence makes sense."
>>> words = s.split(" ")
>>> words
['Pretend', 'this', 'sentence', 'makes', 'sense.'][
>>> "_" .join(words) #join method of obj "_"
'Pretend_this_sentence_makes_sense.'
```

```
>>> s = 'dog'
>>> s.capitalize()
'Dog'
>>> s.upper()
'DOG'
>>> ' hi -- ' .strip(' -')
'hi'
```

<https://docs.python.org/3.7/library/string.html>

# Tuples

```
>>> a = ["apple", "orange", "banana"]
>>> for (index, fruit) in enumerate(a):
... print(str(index) + ": " + fruit)
...
```

**0: apple**

**1: orange**

**2: banana**

```
>>> a = [1, 2, 3]
>>> b = ['a', 'b', 'c', 'd']
>>> list(zip(a, b))
[(1, 'a'), (2, 'b'), (3, 'c')]
```

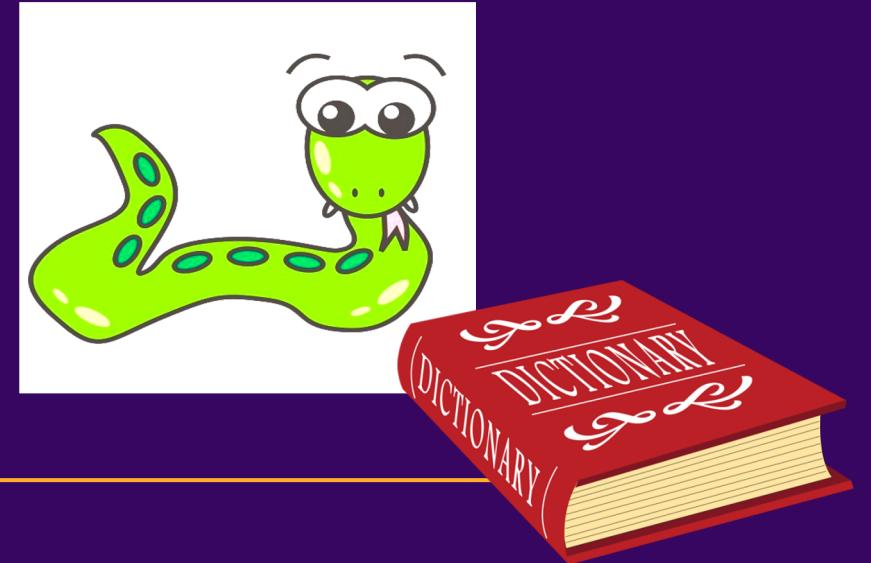
```
>>> list(zip("foo", "bar"))
[('f', 'b'), ('o', 'a'), ('o', 'r')]
```

```
>>> x, y, z = 'a', 'b', 'c'
```

# Dictionaries:

A *mapping* collection type

---



# Dict: Create, Access, Update

- Dictionaries are unordered & work by hashing, so keys must be immutable
- Constant average time add, lookup, update

```
>>> d = { 'user' : 'bozo', 'pswd': 1234}
```

```
>>> d['user']
```

```
'bozo'
```

```
>>> d['bozo']
```

```
Traceback (most recent call last):
```

```
 File "<stdin>", line 1, in <module>
```

```
KeyError: 'bozo'
```

```
>>> d['user'] = 'clown' # Assigning to an existing key replaces its value.
```

```
>>> d
```

```
{'user': 'clown', 'pswd': 1234}
```

# Dict: Useful Methods

```
>>> d = {'user':'bozo', 'p':1234, 'i':34}
>>> d.keys() # List of current keys
dict_keys(['user', 'p', 'i'])
>>> d.values() # List of current values.
dict_values(['bozo', 1234, 34])
>>> d.items() # List of item tuples.
dict_items([('user', 'bozo'), ('p', 1234), ('i', 34)])
```

```
>>> from collections import defaultdict
```

```
>>> d = defaultdict(int)
```

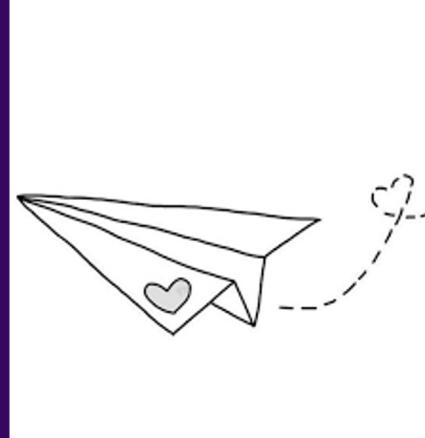
```
>>> d['a']
```

0

- defaultdict automatically initializes nonexistent dictionary values

# For Loops

---



# For Loops

- **for <item> in <collection>:**  
    **<statements>**
- If you've got an existing list, this iterates each item in it.
- You can generate a list with **Range**:

`list(range(5))` returns [0,1,2,3,4]

So we can say:

```
for x in range(5):
 print(x)
```

- **<item> can be more complex than a single variable name.**

```
for (x, y) in [('a',1), ('b',2), ('c',3), ('d',4)]:
 print(x)
```

# List Comprehensions replace loops!

```
nums = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
I want 'n*n' for each 'n' in nums
squares = []
for n in nums:
 for n in nums:
 squares.append(x*x)
print(squares)
```

```
squares = [x*x for x in nums]
print(squares)
```

# List Comprehensions replace loops!

```
>>> li = [3, 6, 2, 7]
>>> [elem * 2 for elem in li]
[6, 12, 4, 14]
```

```
>>> li = [('a', 1), ('b', 2), ('c', 7)]
>>> [n * 3 for (x, n) in li]
[3, 6, 21]
```

# Filtered List Comprehensions

```
>>> li = [3, 6, 2, 7, 1, 9]
>>> [elem * 2 for elem in li if elem > 4]
[12, 14, 18]
```

- Only 6, 7, and 9 satisfy the filter condition.
- So, only 12, 14, and 18 are produced.

# Dictionary, Set Comprehensions

```
lst1 = [('a', 1), ('b', 2), ('c', 'hi')]
lst2 = ['x', 'a', 6]
```

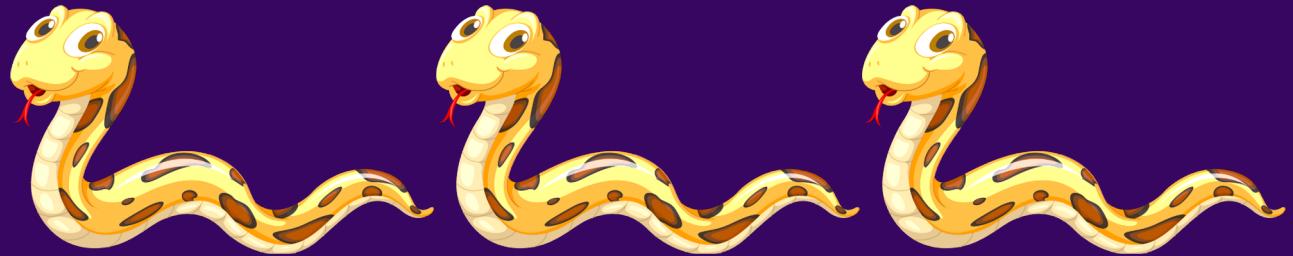
```
d = {k: v for k,v in lst1}
s = {x for x in lst2}
```

```
d = dict() # translation
for k, v in lst1:
 d[k] = v
s = set() # translation
for x in lst:
 s.add(x)
```

```
Both value of d: {'a': 1, 'b': 2, 'c': 'hi'}
Both value of d: {'x', 'a', 6}
```

# Iterators

---



# Iterator Objects

- Iterable objects can be used in a `for` loop because they have an `__iter__` magic method, which converts them to iterator objects:

```
>>> k = [1, 2, 3]
```

```
>>> k.__iter__()
<list_iterator object at 0x104f8ca50>
```

```
>>> iter(k)
<list_iterator object at 0x104f8ca10>
```

# Iterators

- Iterators are objects with a `__next__()` method:

```
>>> i = iter(k)
>>> next(i)
1
>>> i.__next__()
2
>>> i.next()
3
>>> i.next()
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

`StopIteration`

- Python iterators do not have a `hasnext()` method!
- Just catch the `StopIteration` exception

# Iterators: The truth about for... in...

- `for <item> in <iterable>:`  
    `<statements>`
- First line is just syntactic sugar for:
  1. Initialize: Call `<iterable>.__iter__()` to create an *iterator*
- Each iteration:
  2. Call `iterator.__next__()` and bind `<item>`
  - 2a. Catch `StopIteration` exceptions
- To be iterable: has `__iter__` method  
    which returns an iterator obj
- To be iterator: has `__next__` method  
    which throws `StopIteration` when done

# An Iterator Class

```
class Reverse:
 "Iterator for looping over a sequence backwards"
 def __init__(self, data):
 self.data = data
 self.index = len(data)
 def __next__(self):
 if self.index == 0:
 raise StopIteration
 self.index = self.index - 1
 return self.data[self.index]

 def __iter__(self):
 return self

>>> for char in Reverse('spam'):
 print(char)
```

m  
a  
p  
s

# Iterators use memory efficiently

Eg: File Objects

```
>>> for line in open("script.py"): # returns iterator
... print(line.upper())
...
```

IMPORT SYS

PRINT(SYS.PATH)

X = 2

PRINT(2 \*\* 3)

instead of

```
>>> for line in open("script.py").readlines(): #returns list
```

```
... print(line.upper())
```

```
...
```

# Generators

---



# Generators: using yield

- Generators are iterators (with `__next().___` method)
- Creating Generators: `yield`  
Functions that contain the `yield` keyword **automatically** return a generator when called

```
>>> def f(n):
... yield n
... yield n+1

...
>>>
>>> type(f)
<class 'function'>
>>> type(f(5))
<class 'generator'>
>>> [i for i in f(6)]
[6, 7]
```

# Generators: What does yield do?

- Each time we call the `__next__` method of the generator, the method runs until it encounters a `yield` statement, and then it stops and returns the value that was yielded. Next time, it resumes where it left off.

```
>>> gen = f(5) # no need to say f(5).__iter__()
```

```
>>> gen
```

```
<generator object f at 0x1008cc9b0>
```

```
>>> gen.__next__()
```

```
5
```

```
>>> next(gen)
```

```
6
```

```
>>> gen.__next__()
```

**Traceback (most recent call last):**

  File "<stdin>", line 1, in <module>

StopIteration

# Generators

- **xrange(n) vs range(n) in Python 2**

**xrange** acts like a generator

**range(n)** keeps all n values in memory before starting a loop *even if n is huge:*

**for k in range(n)**

**sum(xrange(n))** much faster than **sum(range(n))** for large **n**

- **In Python 3**

**xrange(n)** is removed

**range(n)** acts similar to the old **xrange(n)**

Can use `list()` to get similar behavior as in Python 2

Python 3's range is more powerful than Python 2's xrange

# Generators

- **Benefits of using generators**

- Less code than writing a standard iterator

- Maintains local state automatically

- Values are computed one at a time, as they're needed

- Avoids storing the entire sequence in memory

- Good for aggregating (summing, counting) items. One pass.

- Crucial for infinite sequences

- Bad if you need to inspect the individual values

# Using generators: merging sequences

- Problem: merge two sorted lists, using the output as a stream (i.e. not storing it).

```
def merge(l, r):
 llen, rlen, i, j = len(l), len(r), 0, 0
 while i < llen or j < rlen:
 if j == rlen or (i < llen and l[i] < r[j]):
 yield l[i]
 i += 1
 else:
 yield r[j]
 j += 1
```

# Using generators

```
>>> g = merge([2,4], [1, 3, 5]) #g is an iterator
>>> while True:
... print(g.__next__())
```

...

1

2

3

4

5

**Traceback (most recent call last):**

**File "<stdin>", line 2, in <module>**

**StopIteration**

```
>>> [x for x in merge([1,3,5],[2,4])]
[1, 2, 3, 4, 5]
```

# Generators and exceptions

```
>>> g = merge([2,4], [1, 3, 5])
>>> while True:
... try:
... print(g.__next__())
... except StopIteration:
... print('Done')
... break
...
1
2
3
4
5
Done
```