Guido van Rossum started development on Python in 1989 as a project to keep him busy over the holiday break when his office was closed.

- van Rossum is Python's "Benevolent Dictator for Life"
- Worked for Google from 2005-2013, and currently works at Dropbox
  
  - Both employers had him spend 50% of his time working on Python
Python

- An interpreted language
- Version 2.0 was released in Oct. 2000
- Version 3.0 was released in Dec. 2008
  - 3.0 was a *backwards-incompatible* release
    - Because of this Python 2.7 is still actively used by many
    - 3.6 is the current latest version of Python
    - We will be using 3.x versions of Python in this course!
      - Be aware of the difference when checking online resources and running Python code on your computer!
print("Hello World!")
import random

# basic syntax overview example
r = random.randint(0, 100)
while r < 85:
    if r > 70:
        print(r, ": so close!", sep="")
    elif r > 45:  # yes, the else if syntax is odd...
        print(r, end="")
        print(": Getting there...")
    else:
        print("{{}: Still so far away!}".format(r))

    r = random.randint(0, 100)
print("OUT!")
Like JavaScript, Python is dynamically typed

However, *unlike* JavaScript, Python is *strongly* typed

- 1 + "1" will raise an error
  - 1 + int("1") is fine, as is str(1) + "1"
- 1 == "1" will return false
  - Not the same value, one is a string, one an int
  - Python does not have a === operator
Numerical types

- **int**
- **float**
  - $7 / 2$
    - = 3.5
  - $7 // 2$
    - = 3
  - $7.0 / 2.0$
    - = 3.5
  - $7.0 // 2.0$
    - = 3.0
Numerical operators

- Generally the same as C
  - +, -, *, /, % all work as expected

- Python-specific operators:
  - //
    - Integer division
  - **
    - Exponentiation
Booleans

- True
- False

- Comparison operators:
  - and
  - or
  - not

  (True and False) or (not False) == True
Strings

- Can be "double quoted"
- or 'single quoted'
- or """"TRIPLE QUOTED""
  - ""triple singles also work"
- Plenty of string methods available
  - Note specifically that they can be indexed and sliced
String slicing

s = "slicing is fun!!"

print(s[0])
print(s[2:7])
print(s[-5])
print(s[-5:-2])
print(s[-5: -2])
print(s[11:])
print(s[:7])
print(s[11:])
print(s[-5:])
Functions

def say_hi():
    print("Hi")

def shout(message="Hi"):
    print(message, "!", sep="")

shout()
shout("I love python")
shout(message="And keyword arguments")
Tuples

- Immutable sequences
  - `t = ("CS", 1520, "Farnan")`
  - `t[2] = "Garrison" # ERROR!`

- Note that the () can be omitted in tuple definition:
  - `s = "CS", 1520, "Farnan"
  - `t == s  # True`

- Further tuples can be "unpacked":
  - `a, b, c = t`
    - `a == "CS"`
    - `b == 1520`
    - `c == "Farnan"`
• Note that tuples can be used to make it seem like a function returns multiple values:

def mult_ret():
    return "one", "two", "three"

a, b, c = mult_ret()
Special cases with tuples

- How do you create an empty tuple?
- How about a tuple with only 1 item in it?
Lists

- Mutable sequences
  - \[ l = [1, 2, 5] \]
  - \[ l[0] = 0 \]
  - \[ l.append(3) \]
  - \[ if 3 in l: \]
    - \[ print(3, "is in", l) \]
Key/value stores

- \( d = \{\text{"Farnan":[1520], \text{"Garrison":[8]}\} \)

  \( d[\text{"Ramirez"] = 1501 \)

  \( d[\text{"Garrison"] = \text{"0008"} \)

- \( \text{"0008" in } d \)

- \( \text{"Garrison" in } d \)
Sets

- Unordered collections with no duplicate elements
  - \( s = \{1, 2, 2, 3, 3, 3\} \)
  - \texttt{print(s)}
    - # prints: \( \{1, 2, 3\} \)
Collection function examples

- `set()`
  - Produces an empty set: `{}`
- `set([1, 2, 2, 3, 3, 3])`
  - Produces `{1, 2, 3}`
- `list(range(10))`
  - Produces `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`
- `dict([("k1", "v1"), ("k2", "v2"), ("k3", "v3")])`
  - Produces `{ 'k1': 'v1', 'k2': 'v2', 'k3': 'v3' }`
Looping

- Already saw a while example...
- **for** is quite interesting, though:
  - ```python
crazy_list = ["a", 1, 1.0, "d"]
  for item in crazy_list:
      print(item)
  ```
  ```python
  for i in range(len(crazy_list)):
      print(crazy_list[i])
  ```
Can loop over dictionaries as well:

- `crazy_dict = {1:"one", 2:"two", 3:"three"}`
- `for k in crazy_dict:
  print(k, crazy_dict[k])`
- `for k, v in crazy_dict.items():
  print(k, v)`
List comprehensions

- Succinct way to initialize lists:
  - `squares = [x**2 for x in range(10)]`
  - `names = ["NICK", "FARNAN"]`
  - `low = [n.lower() for n in names if n == "NICK"]`
Both lists, range objects, etc. are *iterable*

- Meaning that an *iterator* can be created for either type
  - Iterators must implement the method `__next__()`
    - Successive calls to `__next__()` will iterate through the items in the collection
    - When no more items remain in the collection, all future calls to `__next__()` should raise a *StopIteration* exception
Exceptions and try statements

```python
try:
    result = x / y
except ZeroDivisionError:
    print("division by zero!")
else:
    print("result is", result)
finally:
    print("executing finally clause")

try:
    raise Exception("foo", "bar")
except Exception as e:
    print(e)
    print(type(e))
    print(e.args)
```
temp_iter = iter(crazy_list)
while True:
    try:
        item = temp_iter.__next__()
    except StopIteration:
        break
    print(item)
Generators

- Functions that use the yield keyword to return values in order to create iterators
  - State is maintained between function calls to the generator

```python
def enum(seq):
    n = 0
    for i in seq:
        yield n, i
        n += 1

def fibonacci():
    i = j = 1
    while True:
        r, i, j = i, j, i + j
        yield r
```
def plus_one(original_function):
    def new_function(x, y):
        return original_function(x, y) + 1
    return new_function

@plus_one
def add(x, y):
    return x + y
```python
outf = open("example.txt", "w")
for i in range(10):
    outf.write(str(i) + "\n")
outf.close()

inf = open("example.txt")
for line in inf:
    print(line.strip())
inf.close()
```
with open("example.txt") as inf:
    for line in inf:
        print(line.strip())
from contextlib import contextmanager

@contextmanager
def tag(name):
    print("<{}>".format(name))
    yield
    print("</{}>".format(name))

with tag("h1"):
    print("foo")
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def display(self):
        print("Name:", self.name)
        print("Age:", self.age)
        print()
class Student(Person):
    def __init__(self, name, age):
        super().__init__(name, age)

        self.classes = []

    def add_class(self, new):
        self.classes.append(new)

    def display(self):
        super().display()
        print("Classes:", self.classes)
        print()
class Dog:
    tricks = []
    def __init__(self, name):
        self.name = name
    def add_trick(self, trick):
        self.tricks.append(trick)
f = Dog("Fido")
b = Dog("Buddy")
f.add_trick("roll over")
b.add_trick("play dead")
print(f.tricks)
wow

such instance variables

class Dog:
    def __init__(self, name):
        self.tricks = []
        self.name = name
    def add_trick(self, trick):
        self.tricks.append(trick)
Any Python file is a module that can be imported into other Python modules with `import`.

Let `a.py` contain:

- `def print_n():`
  - `for i in range(10):`
    - `print(i)`
- `def print_l():`
  - `for l in ["a", "b", "c"]:`
    - `print(l)`

Can then (in other files):

- `import a`
  - `a.print_n()`
- `from a import print_l`
  - `print_l()`
Consider:

- Running `python a.py` from the command line
- Having `import a` in another Python file

How can we have the former produce output while still being able to use the latter to pull in definitions??

- Both will evaluate each line of a.py
- However, `python a.py` will have `__name__` set to "`__main__`"
- Hence, we can choose what to do if run as a script:
  - At the end of a.py, have:
    - `if __name__ == "__main__":`
      - `print("Producing output!")`
Handy build in functions:

- `len()`
- `sorted()`
- `min()`, `max()`
- `int()`, `str()`, `bool()`, `float()`
- `repr()`
- `type()`
- `help()`
- ...

To wrap up...

- This is only a brief introduction to Python
- There are *many* topics that we neared but did not touch upon
  - E.g.,
    - Multiple inheritance
    - Packages
    - Protocols
      - E.g.,
        - Containers
        - Iterators
        - Context Managers
      - ...