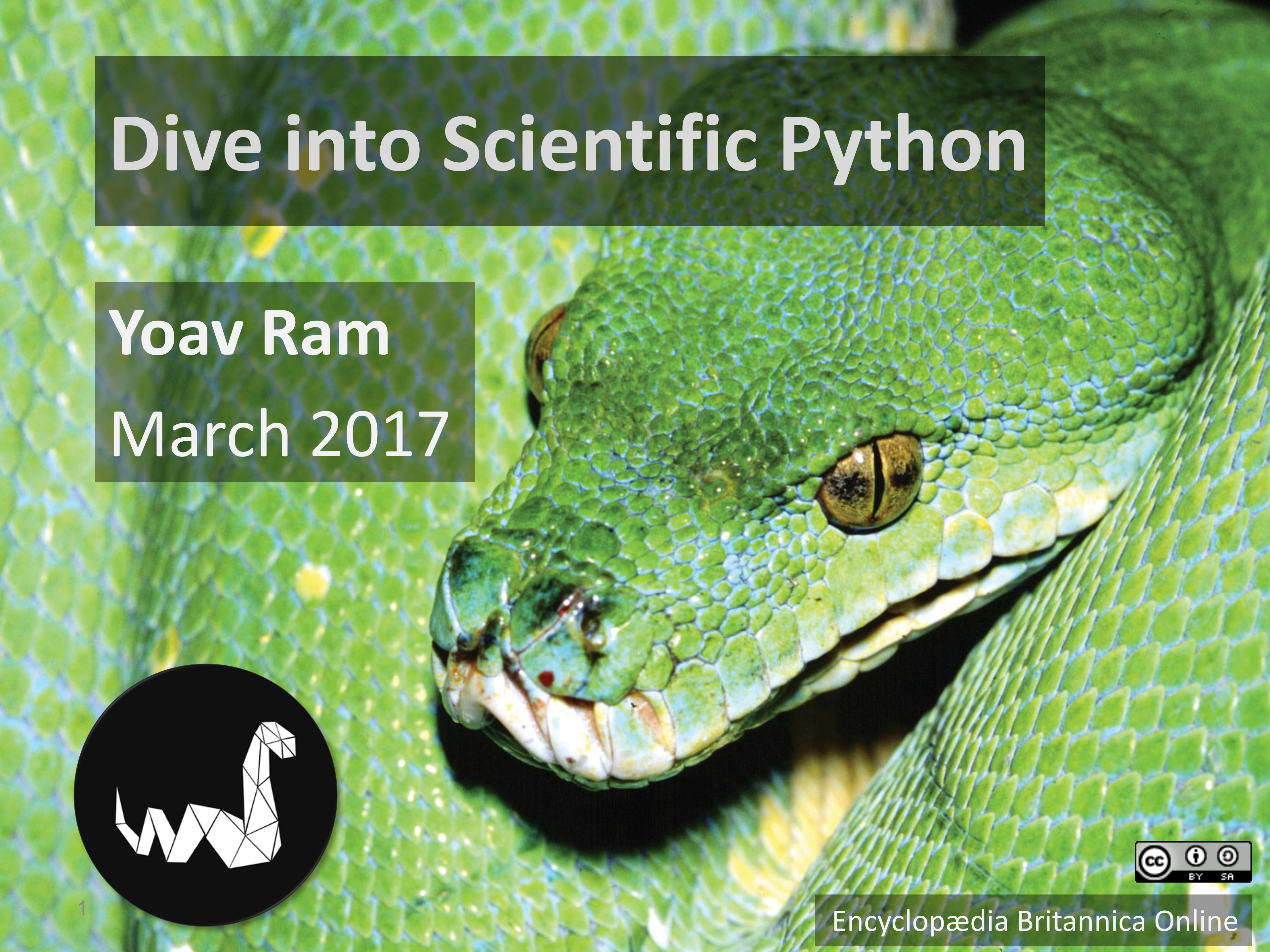


Dive into Scientific Python

Yoav Ram

March 2017

Encyclopædia Britannica Online

Who I am

- **Yoav Ram** @yoavram
- Postdoc at Stanford University
- PhD in BioMath from Tel-Aviv University
- Using Python since 2002
- Using & teaching **Scientific Python** since 2011
- Python training for engineers & data scientists

Presentation & source code on GitHub:
http://**git.io/14032017**

Why Python?

for scientific computing…

[I used Matlab. Now I use Python. by Steve Tjoa]
[Why use Python for scientific computing?]

Python is Free

Gratis: Free as in Beer

- MATLAB is **expensive** (as of Feb 2017)
  - Individuals: $2,350
  - Academia: $550
  - Personal: $85
  - Student: $29-55
  - Batteries (toolboxes…) not included
- Python is totally **free**
  - Batteries included (NumPy, SciPy…)
- R is also **free**

MathWork Pricing

Libre: Free as in Speech

- MATLAB source code is **closed** and proprietary
  - You cannot **see** the code
  - You cannot **change** the code
  - You can participate in the discussion as a **client**
- Python source code is **open**
  - You can **see**, you can **change,** you can **contribute** code and documentation (python, numpy)
  - You can participate in the discussion as a **peer** (python, numpy)
- R is also **open**

Python is a general-purpose language

R and MATLAB are used primarily for scientific computing

Python is used for:

- Scientific computing
- Enterprise software
- Web design
- Back-end
- Front-end
- Everything in between

Python is used at

Google, Rackspace, Microsoft, Intel, Walt Disney, MailChimp, twilio, Bank of America, Facebook, Instagram, HP, Linkedin, Elastic, Mozilla, YouTube, ILM, Thawte, CERN, Yahoo!, NASA, Trac, Civilization IV, reddit, LucasFilms, D-Link, Phillips, AstraZeneca, KLA-Tencor, **Nerua**

https://us.pycon.org/2016/sponsors/
https://www.python.org/about/quotes/
https://en.wikipedia.org/wiki/Python_%28programming_language%29#Use
https://en.wikipedia.org/wiki/List_of_Python_software
https://www.python.org/about/success/

Python is portable

More or less same code runs on Windows, Linux, macOS, and any platform with a Python interpreter

[Python for "other" platforms]

Python syntax is beautiful

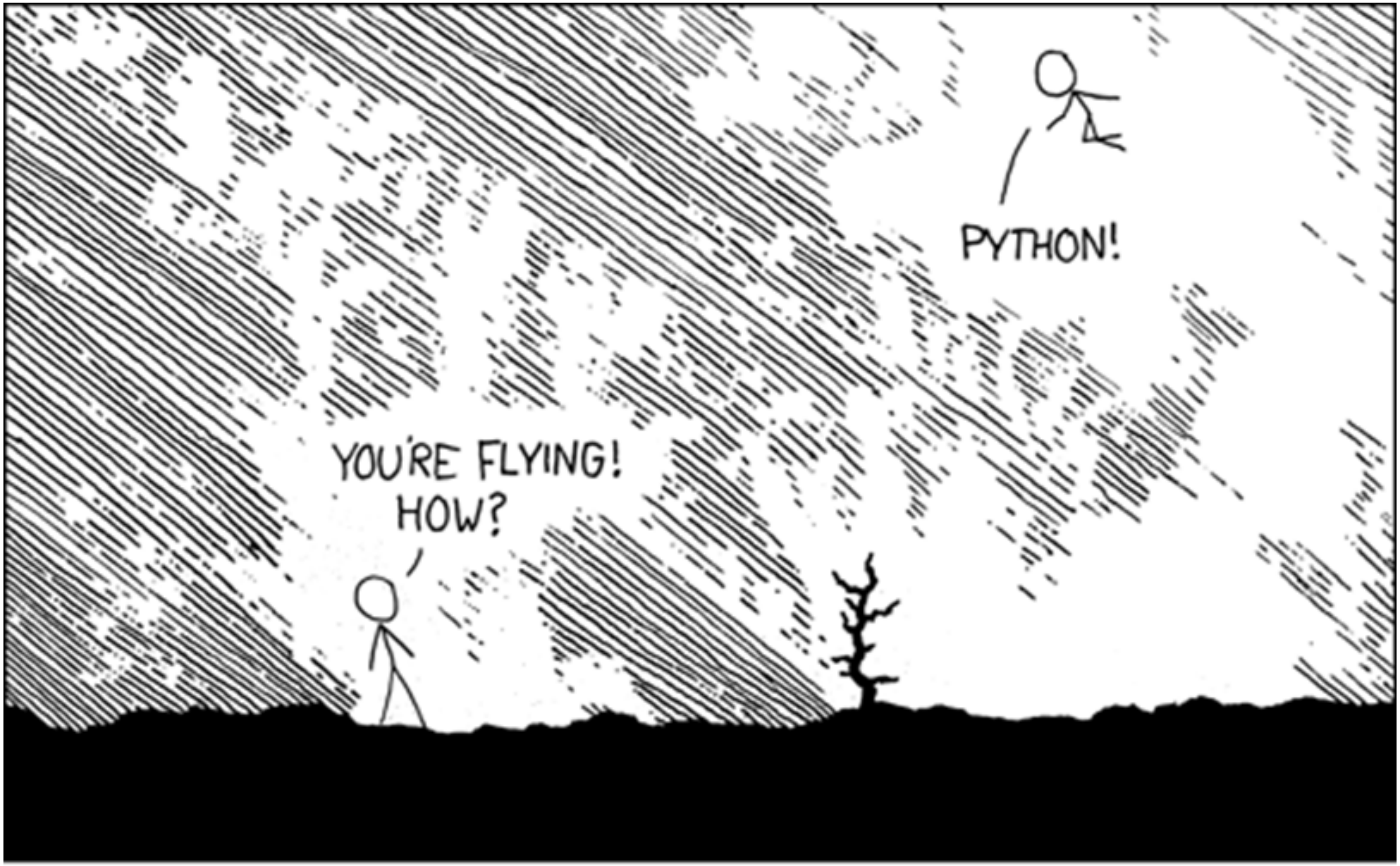
Although beauty is in the eyes of the beholder

Python is inherently object-oriented

Almost everything is an object

Python is high level, easy to learn, and fast to develop

So is MATLAB, Ruby, R…



XKCD 353

Python is fast enough

Written in C

Easy to wrap more C

Easy to parallelize

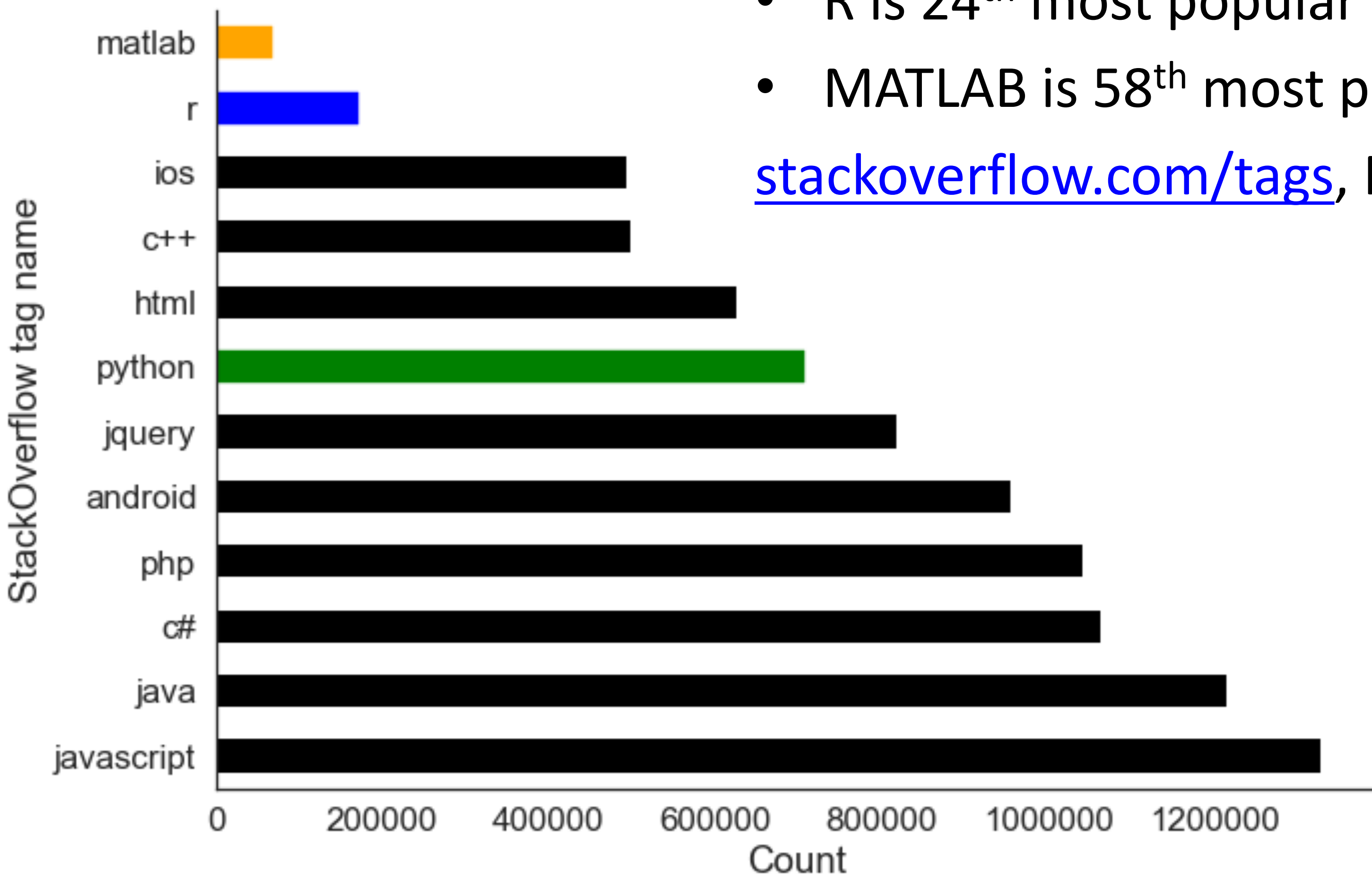
Benchmark Game | NumFocus Benchmarks

Python is popular and has a great community

Popularity

- Python is 7th most popular tag on StackOverflow
- R is 24th most popular tag
- MATLAB is 58th most popular tag

stackoverflow.com/tags, Feb 2017

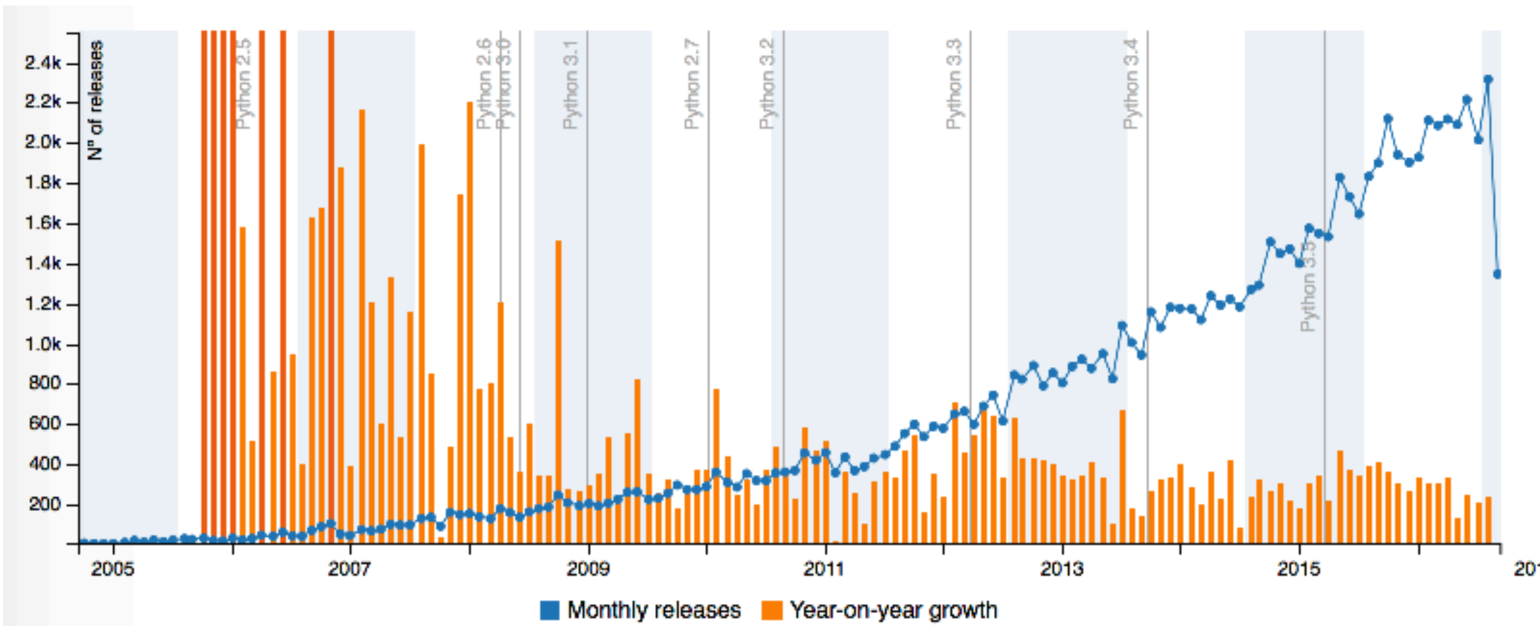


Active community

- 3rd most active repositories on *GitHub* after *Java* (incl. *Android*) and *JavaScript* (incl. *node.js*)
- ~4.8-fold more than R (12th)
- ~27-fold more than MATLAB (24th)
- As of Feb 2017
- See breakdown at githut

Python has a lot of great libraries

Many new libraries released every month



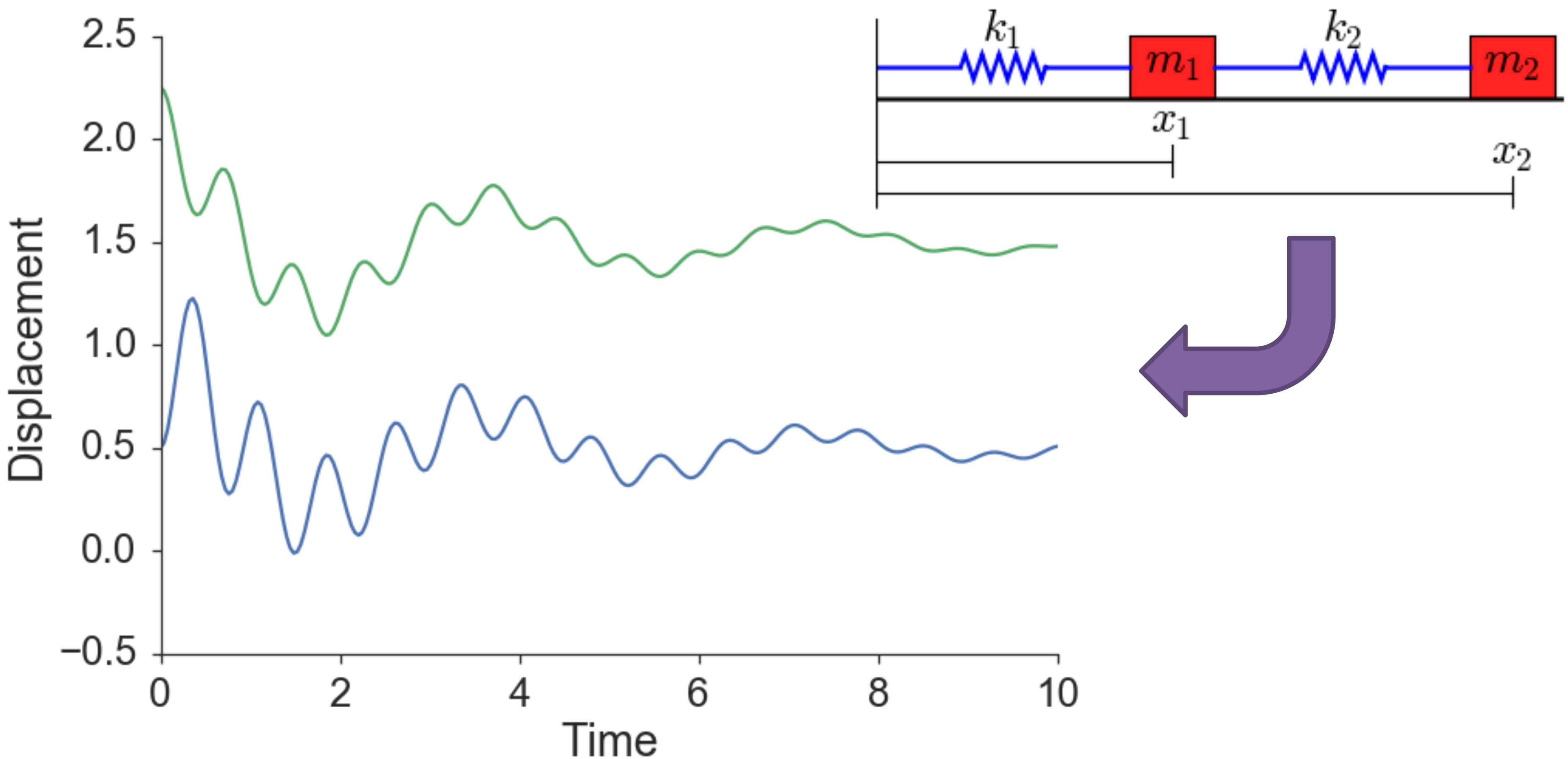
During 2016 >2,000 new packages released every month.
See more stats at PyGarden/stats.

Python can do nearly everything MATLAB and R can do

With libraries like NumPy, SciPy, Matplotlib, IPython/Jupyter,

Scikit-image, Scikit-learn, and more

Differential equations

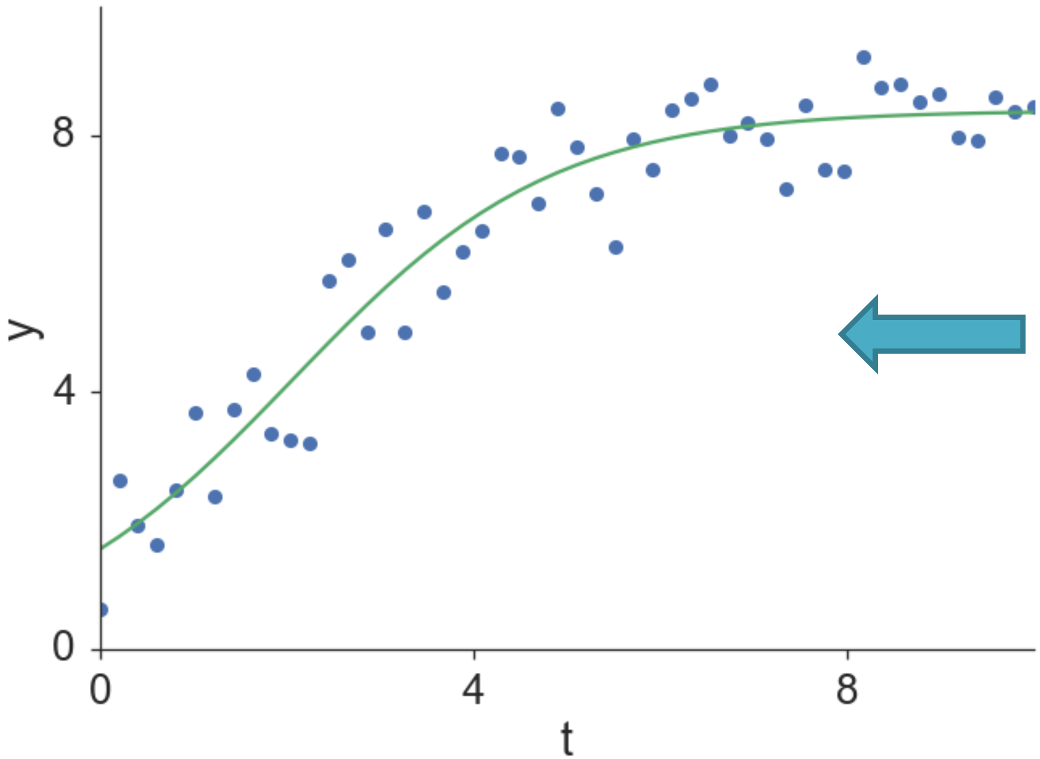



```
x = scipy.integrate.odeint(f, w0, t, …)
plot(t, x[:, 0])
plot(t, x[:, 1])
```

Model fitting

```
params, cov = scipy.optimize.curve_fit(
      f=logistic, xdata=t, ydata=y, p0=(1, 10, 1))
```

N0=1.512, K=8.462, r=0.758



$$N(t) = \frac{K}{1 - \left(1 - \frac{K}{N(0)}\right)e^{-rt}}$$

Optimization

```
res = scipy.optimize.minimize_scalar(
          f, method="bounded", bounds=[8, 16])
```

fun: -0.23330441717143405

message: 'Solution found.'

nfev: 9

status: 0

success: True

x: 11.706005

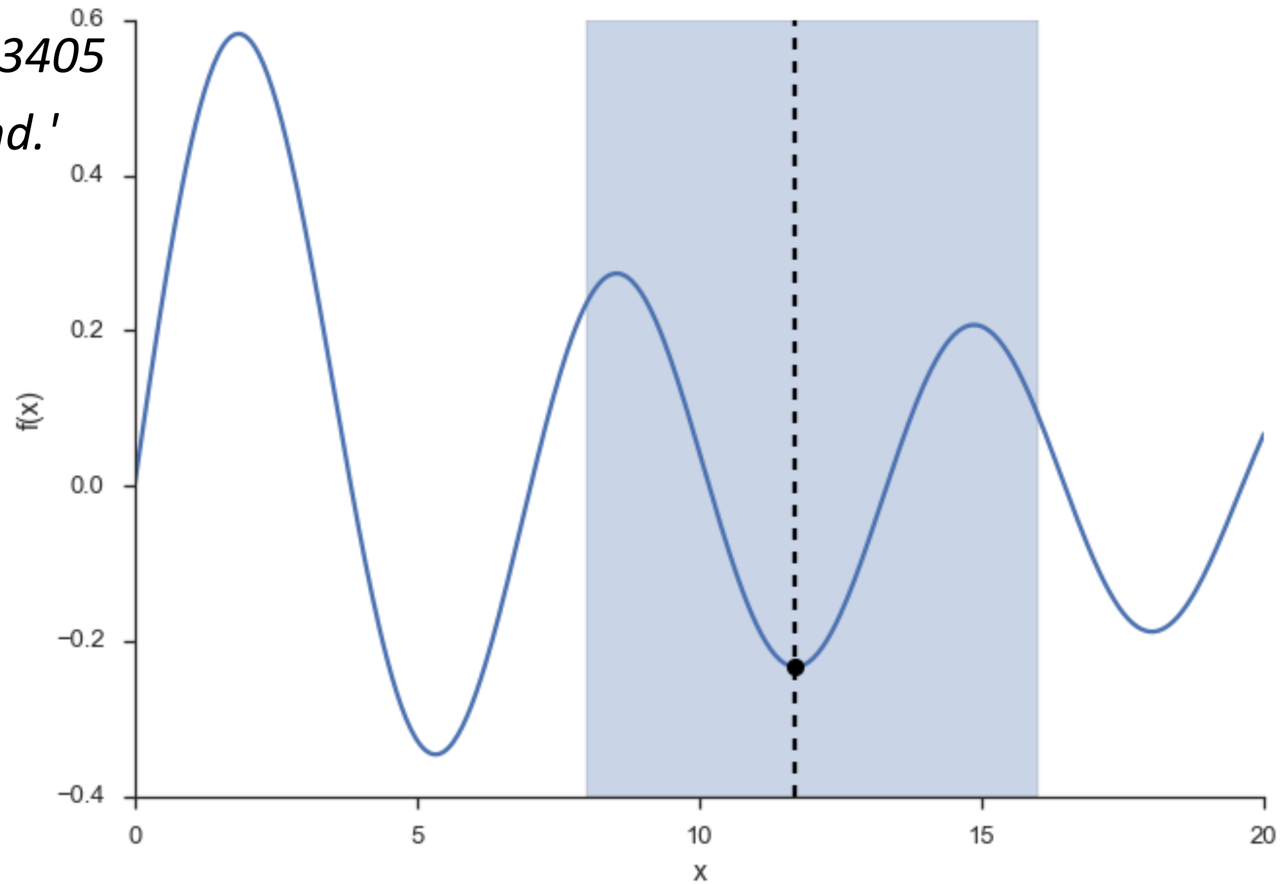
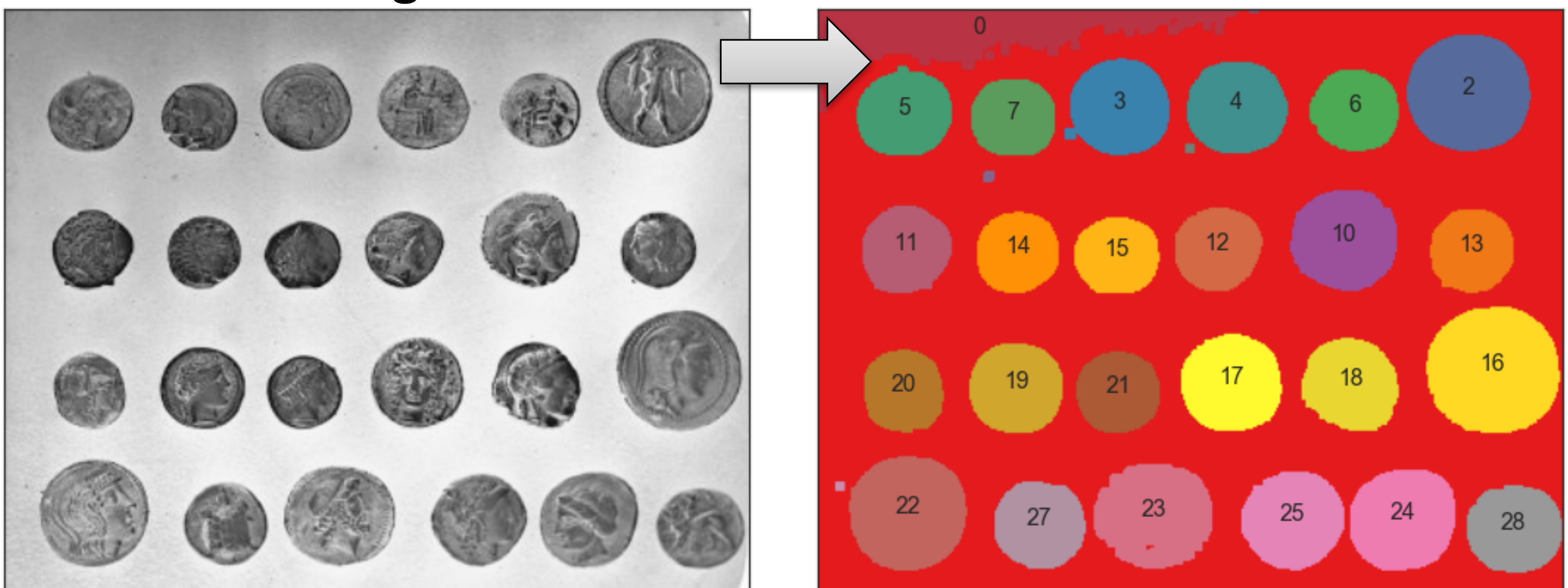


Image analysis

```
segmented = image > threshold
dilated = scipy.ndimage.generic_filter(segmented, max)
labels = skimage.measure.label(dilated)
```

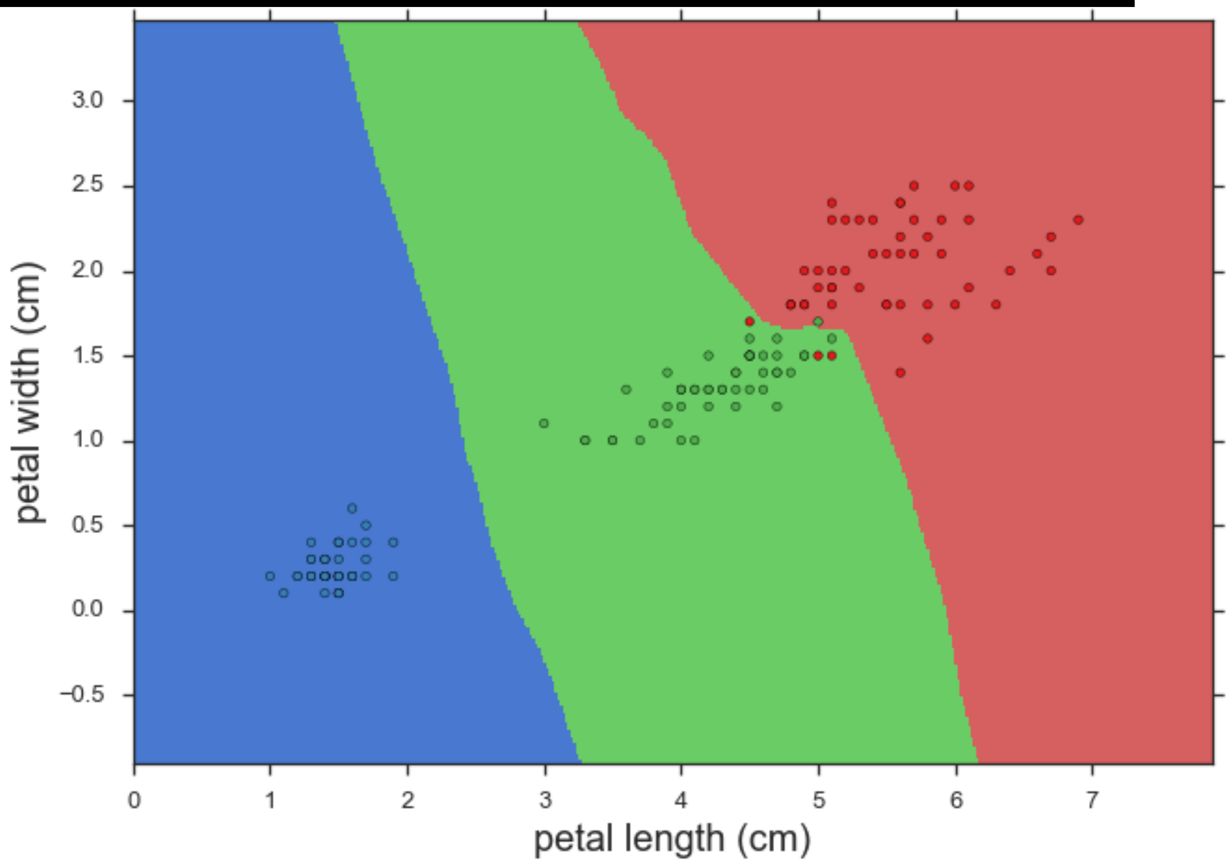
image labels



Machine learning

```
knn = sklearn.neighbors.KNeighborsClassifier()
knn.fit(X_train, y_train)
knn.predict(X_test)
```

Accuracy: 0.9



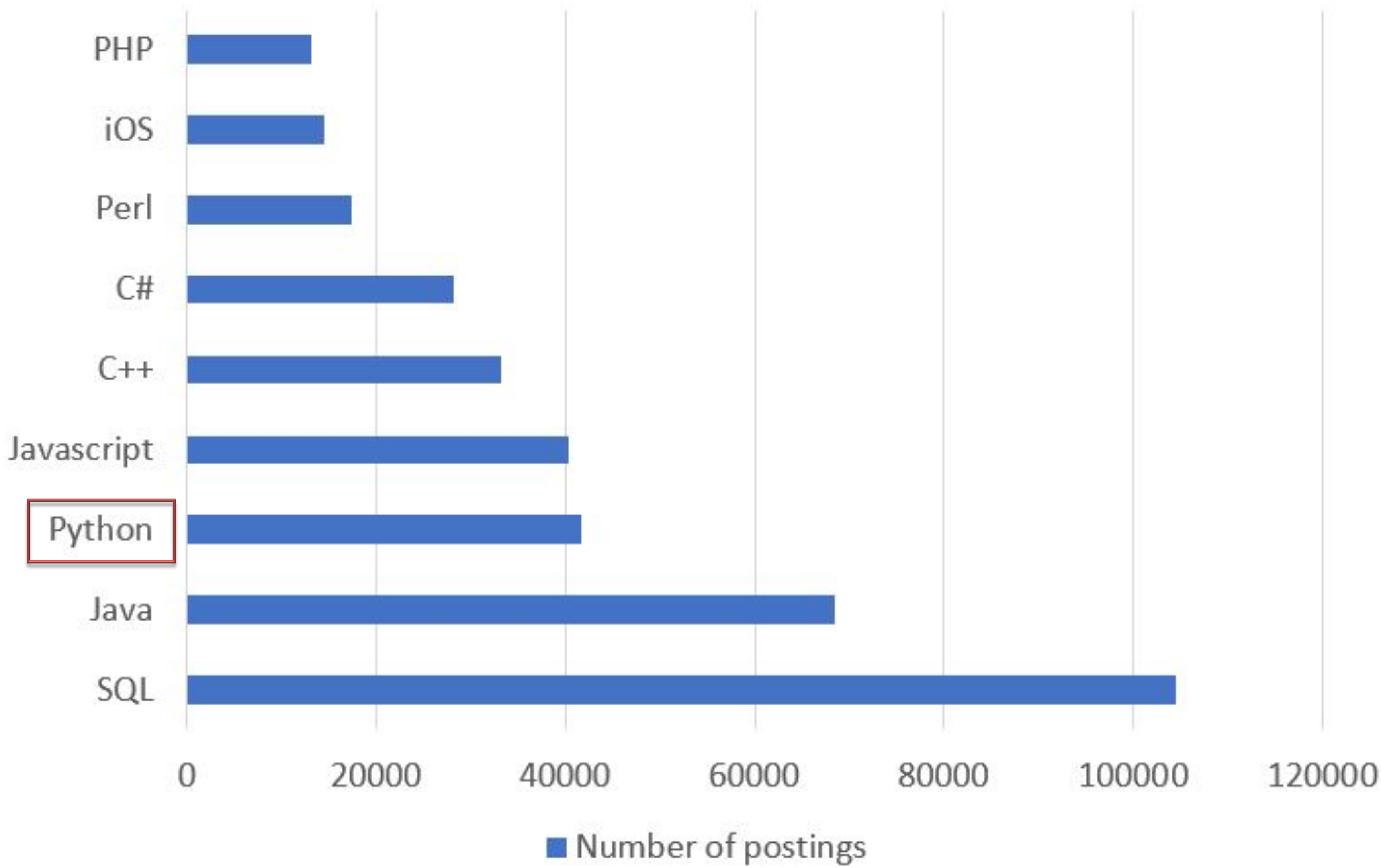
Deep learning

```
with tensorflow.Session() as s:
  readout = s.graph.get_tensor_by_name('softmax:0')
  predictions = s.run(readout, {'Image': image_data})
pred_id = predictions.argmax()
Label = node_lookup.id_to_string(pred_id)
score = predictions[pred_id]
```

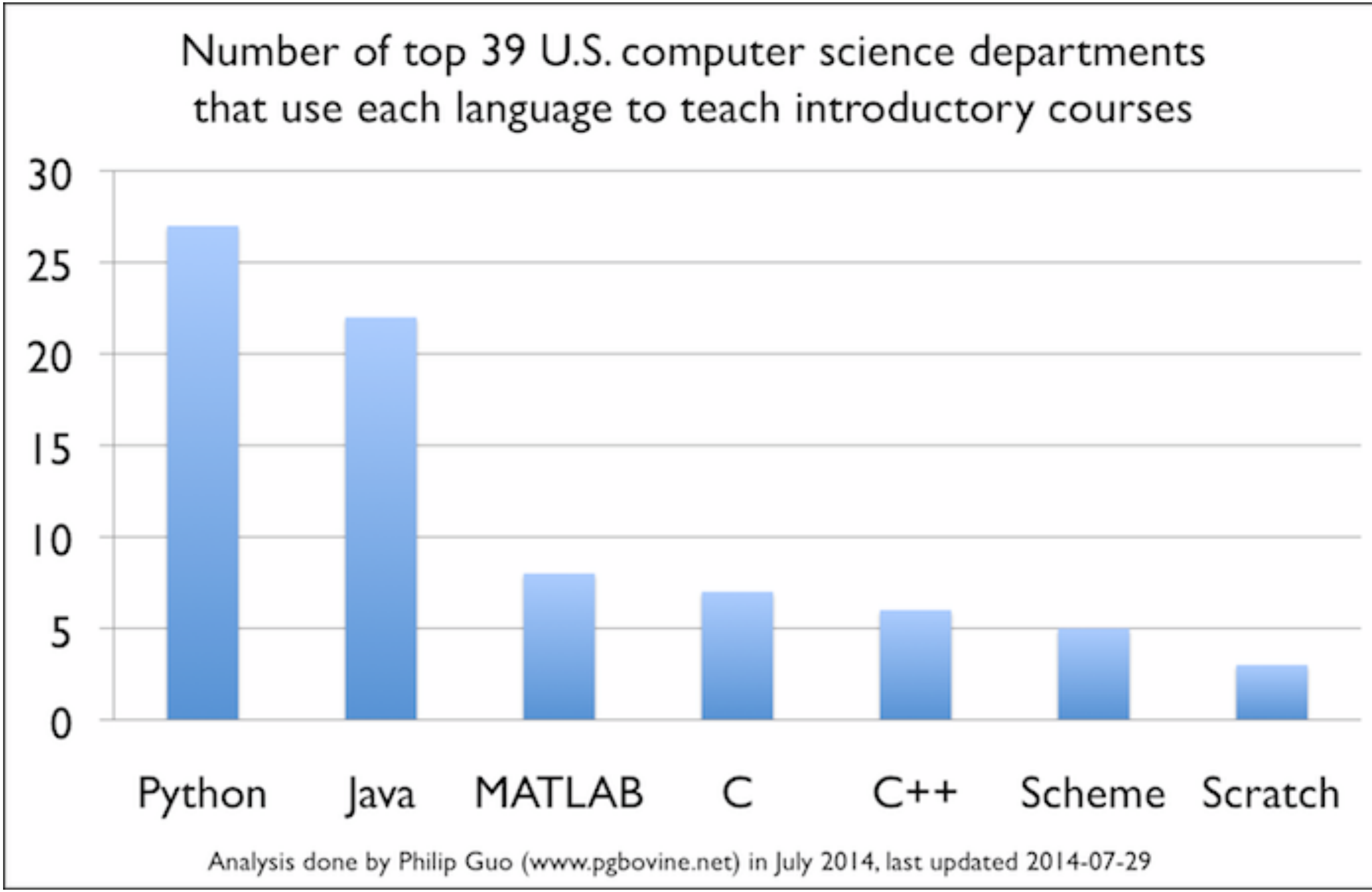
basketball (score = 0.98201)

Demand & supply of Python programmers is high

Number of Indeed Job Postings by Programming Language (Feb 2, 2017)



[Coding Dojo]



Phillip Gou @ CACM

First language at Israeli universities

- **TAU**: CS & Engineering use Python
- **Technion**: CS use C, some courses in Python
- **HUJI**: CS & Humanities, use Python
- **BGU**: CS use Java, Engineering use C

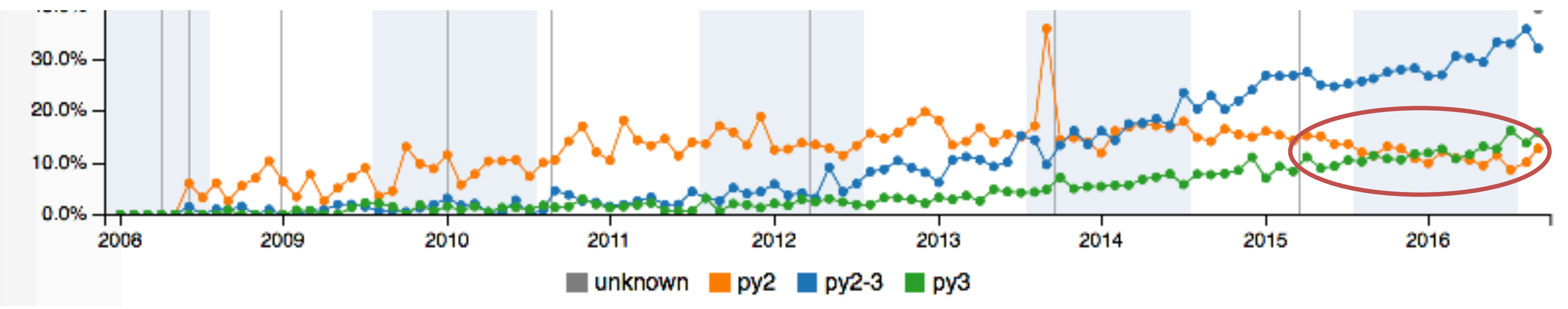
History of Python

- Developed in 1989-91 by **Guido van Rossum** in the Netherlands
- Python 2.0 released Oct 2000 (support ends 2020)
- Python 3.0 released Dec 2008
- Python 3.6 released Dec 2016
- Python 3 is **widely used**

2 vs 3

If you use Python 2.x:

- 2012 called, they want their *print* back
- Seriously, consider moving to 3.x ASAP
- But at least 3.4
- See www.python3statement.org



http://pygarden.com/stats

Scientific Python in action:
Theoretical Evolution

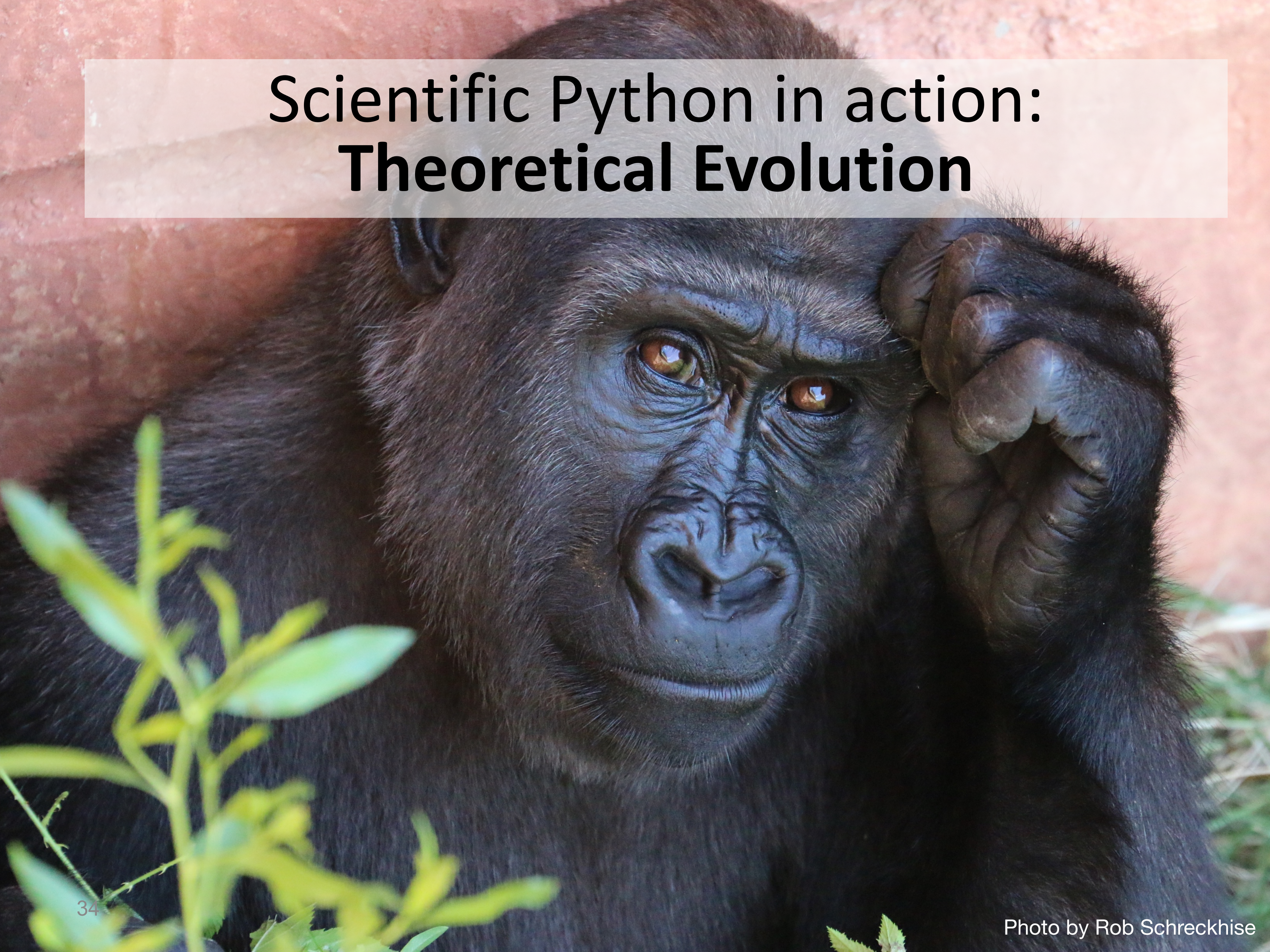


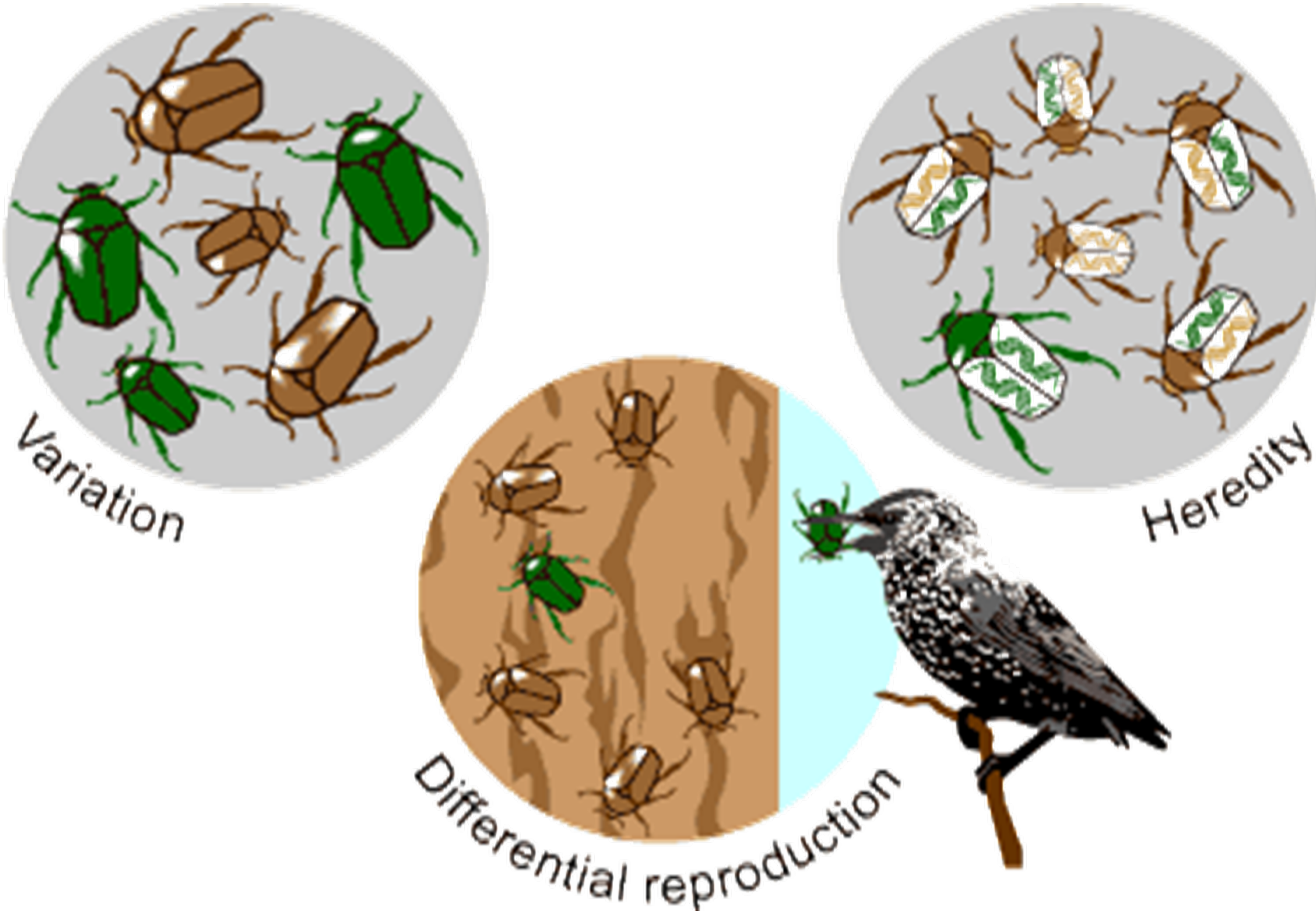
Photo by Rob Schreckhise

Scientific Python in action: **Theoretical Evolution**

- Formally this field is Population Genetics
- Study **changes in frequency** of gene variants within populations
- Focus on two forces:
  - **Natural selection**
  - **Random genetic drift**
- Methods from applied math, statistics, CS, theoretical physics

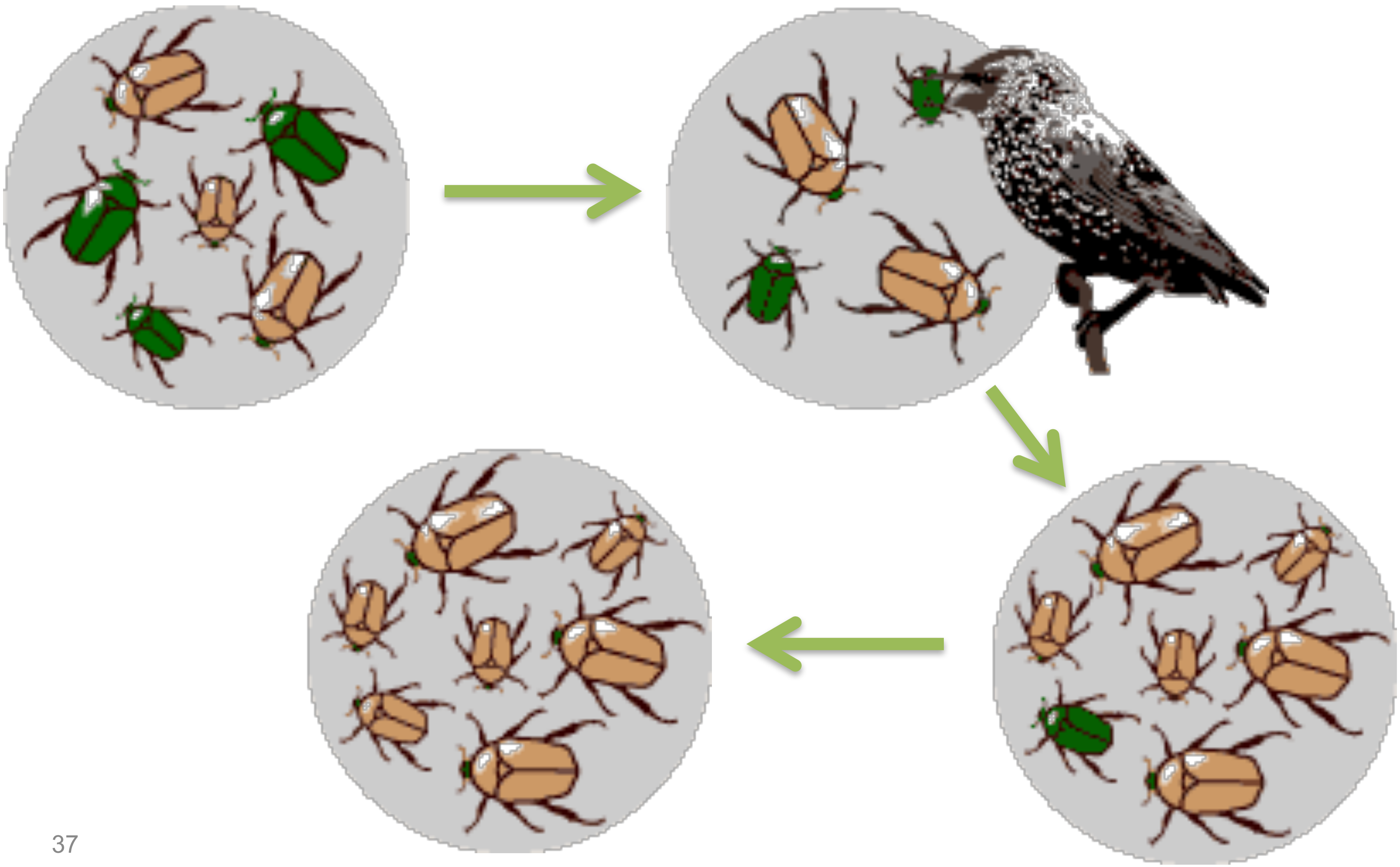
Population genetics

Evolution



University of California Museum of Paleontology
[Understanding Evolution]

Natural Selection



Random Genetic Drift

Wright-Fisher Model

Standard model for change in frequency of gene variants

R.A. Fisher
1890-1962
UK & Australia

Sewall Wright
1889-1988
USA

Wright-Fisher Model

Standard model for change in frequency of gene variants

Two gene variants: **0** and **1**.

Number of individuals with each variant is $\mathbf{n_0}$ and $\mathbf{n_1}$.

Total population size is $\mathbf{N = n_0 + n_1}$.

Frequency of each variant is $\mathbf{p_0=n_0/N}$ and $\mathbf{p_1=n_1/N}$.

Wright-Fisher Model

Assume that variant **1** is **favored by selection** due to better survival or reproduction.

The frequency of variant **1** after the effect of selection natural (**p_1**) is:

$$p_1 = \frac{n_1 \cdot (1 + s)}{n_0 + n_1 \cdot (1 + s)}$$

s is a selection coefficient, representing **how much variant 1 is favored over variant 0.**

Wright-Fisher Model

Random genetic drift accounts for the effect of **random sampling**.

Due to genetic drift, the number of individuals with variant **1** in the next generation (**n'_1**) is:

$$n'_1 \sim \mathrm{Binomial}(N, p_1)$$

The **Binomial distribution** is the distribution of the number of successes in **N** independent trials with probability of success **p_1**.

Fixation Probability

Assume a single copy variant **1** in a population of size **N**.

What is the probability that variant **1** will **take over the population rather than go extinct**?

NumPy

The fundamental package for **scientific computing with Python**:

- N-dimensional arrays
- **Random number generators**
- Array functions
- Broadcasting
- Tools for integrating C/C++ and Fortran code
- Linear algebra
- Fourier transform

numpy.org

Into the code

Death to the Stock Photo

Natural Selection

$$p_1 = \frac{n_1 \cdot (1+s)}{n_0 + n_1 \cdot (1+s)}$$

Random drift

$$n_1' \sim \mathrm{Binomial}(N, p_1)$$

```
from numpy.random import binomial
n1 = 1

while 0 < n1 < N:
  n0 = N - n1
  p1 = n1*(1+s) / (n0 + n1*(1+s))
  n1 = binomial(N, p1)


fixation = n1 == N
```

Import a binomial random number generator from NumPy

Natural Selection

$$p_1 = \frac{n_1 \cdot (1+s)}{n_0 + n_1 \cdot (1+s)}$$

Random drift

$$n_1' \sim \mathrm{Binomial}(N, p_1)$$

```
from numpy.random import binomial
n1 = 1

while 0 < n1 < N:
    n0 = N - n1
    p1 = n1*(1+s) / (n0 + n1*(1+s))
    n1 = binomial(N, p1)

fixation = n1 == N
```

Start with a single copy of variant 1

Natural Selection

$$p_1 = \frac{n_1 \cdot (1+s)}{n_0 + n_1 \cdot (1+s)}$$

Random drift

$$n_1' \sim \mathrm{Binomial}(N, p_1)$$

```
from numpy.random import binomial
n1 = 1
```

Until number of individuals with variant 1 is 0 or N: extinction or fixation

```
while 0 < n1 < N:
  n0 = N - n1
  p1 = n1*(1+s) / (n0 + n1*(1+s))
  n1 = binomial(N, p1)


fixation = n1 == N
```

Natural Selection

$$p_1 = \frac{n_1 \cdot (1+s)}{n_0 + n_1 \cdot (1+s)}$$

Random drift

$$n_1' \sim \mathrm{Binomial}(N, p_1)$$

```
from numpy.random import binomial
n1 = 1

while 0 < n1 < N:
  n0 = N - n1
  p1 = n1*(1+s) / (n0 + n1*(1+s))
  n1 = binomial(N, p1)

fixation = n1 == N
```

The frequency of variant 1 after selection is p1

Natural Selection

$$p_1 = \frac{n_1 \cdot (1+s)}{n_0 + n_1 \cdot (1+s)}$$

Random drift

$$n_1' \sim \mathrm{Binomial}(N, p_1)$$

```
from numpy.random import binomial
n1 = 1

while 0 < n1 < N:
  n0 = N - n1
  p1 = n1*(1+s) / (n0 + n1*(1+s))
  n1 = binomial(N, p1)

fixation = n1 == N
```

Due to genetic drift, the number of individuals with variant 1 in the next generation is n1

Natural Selection

$$p_1 = \frac{n_1 \cdot (1+s)}{n_0 + n_1 \cdot (1+s)}$$

Random drift

$$n_1' \sim \mathrm{Binomial}(N, p_1)$$

```
from numpy.random import binomial
n1 = 1
```

Fixation: n1 **equals** N
Extinction: n1 **equals** 0

```
while 0 < n1 < N:
  n0 = N - n1
  p1 = n1*(1+s) / (n0 + n1*(1+s))
  n1 = binomial(N, p1)

fixation = n1 == N
```

NumPy vs. Pure Python

NumPy is useful for random number generation:

```
n1 = binomial(N, p1)
```

Pure Python version would replace this with:

```
from random import random
rands = (random() for _ in range(N))
n1 = sum(1
         for r in rands
         if r < p1)
```

`random` is a standard library module

NumPy vs. Pure Python

```
%timeit simulation(N=1000, s=0.1)
%timeit simulation(N=1000000, s=0.01)
```

Pure Python version:

100 loops, best of 3: **6.42 ms** per loop

1 loop, best of 3: **528 ms** per loop

NumPy version:

10000 loops, best of 3: **150 µs** per loop **x42 faster**

1000 loops, best of 3: **313 µs** per loop

x1680 faster!


Can we do it
~~better~~ **faster**?
Photo by Malene Thyssen

- **Optimizing compiler**
- Declare the **static type** of variables
- Makes **writing C extensions** for Python as easy as Python itself
- Foreign function interface for invoking C/C++ routines

http://cython.org

```
def simulation(np.uint64_t N,
               np.float64_t s):
    cdef np.uint64_t n1 = 1
    cdef np.uint64_t n0
    cdef np.float64_t p

    while 0 < n1 < N:
        n0 = N - n1
        p1 = n1 * (1 + s) / (n0 + n1 * (1 + s))
        n1 = np.random.binomial(N, p1)

    return n1 == N
```

```
%timeit simulation(N=1000, s=0.1)
%timeit simulation(N=1000000, s=0.01)
```

Cython vs. NumPy:

10000 loops, best of 3: **87.8 µs** per loop **x2 faster**

10000 loops, best of 3: **177 µs** per loop **x1.75 faster**

To approximate the fixation probability we need to run **many simulations**. Thousands.

In principle, the standard error of our approximation decreases with the square root of the number of simulations: $\mathrm{SEM} \sim 1/\sqrt{m}$

Death to the Stock Photo

Multiple simulations: `for` loop

```
fixations = [
    simulation(N, s)
    for _ in range(1000)
]
```

Multiple simulations: `for` loop

```
fixations = [
    simulation(N, s)
    for _ in range(1000)
]
fixations
```

```
[False, True, False, ..., False, False]
```

```
sum(fixations) / len(fixations)
```

```
0.195
```

Multiple simulations: `for` loop

```
%%timeit
fixations = [
    simulation(N, s)
    for _ in range(1000)
]
```

1 loop, best of 3: **8.05 s** per loop

Multiple simulations: NumPy

```
def simulation(N, s, repetitions):
    n1 = np.ones(repetitions)
    update = np.array([True] * repetitions)

    while update.any():
        p1 = n1 * (1 + s) / (N + n1 * s)
        n1[update] = binomial(N, p1[update])
        update = (n1 > 0) & (n1 < N)

    return n1 == N
```

Initialize multiple simulations

Multiple simulations: NumPy

```
def simulation(N, s, repetitions):
  n1 = np.ones(repetitions)
  update = np.array([True] * repetitions)

  while update.any():
    p1 = n1 * (1 + s) / (N + n1 * s)
    n1[update] = binomial(N, p1[update])
    update = (n1 > 0) & (n1 < N)

  return n1 == N
```

Natural selection:
`n1` **is an array so operations are element-wise**

Multiple simulations: NumPy

```
def simulation(N, s, repetitions):
    n1 = np.ones(repetitions)
    update = np.array([True] * repetitions)

    while update.any():
        p1 = n1 * (1 + s) / (N + n1 * s)
        n1[update] = binomial(N, p1[update])
        update = (n1 > 0) & (n1 < N)

    return n1 == N
```

Genetic drift:
`p1` **is an array so** `binomial(N, p1)` **draws from multiple distributions**

Multiple simulations: NumPy

```
def simulation(N, s, repetitions):
  n1 = np.ones(repetitions)
  update = np.array([True] * repetitions)

  while update.any():
    p1 = n1 * (1 + s) / (N + n1 * s)
    n1[update] = binomial(N, p1[update])
    update = (n1 > 0) & (n1 < N)

  return n1 == N
```

`update` **follows the simulations that didn't finish yet**

Multiple simulations: NumPy

```
def simulation(N, s, repetitions):
    n1 = np.ones(repetitions)
    update = np.array([True] * repetitions)

    while update.any():
        p1 = n1 * (1 + s) / (N + n1 * s)
        n1[update] = binomial(N, p1[update])
        update = (n1 > 0) & (n1 < N)

    return n1 == N
```

`update` **follows the simulations that didn't finish yet**

Multiple simulations: NumPy

```
def simulation(N, s, repetitions):
  n1 = np.ones(repetitions)
  update = np.array([True] * repetitions)

  while update.any():
    p1 = n1 * (1 + s) / (N + n1 * s)
    n1[update] = binomial(N, p1[update])
    update = (n1 > 0) & (n1 < N)

  return n1 == N
```

`result` **is array of Booleans: for each simulation, did variant `1` fix?**

Multiple simulations: NumPy

```
%timeit simulation(N=1000, s=0.1)
```

10 loops, best of 3: **25.2 ms** per loop

x320 faster

Fixation probability as a function of N

```
Nrange = np.logspace(1, 6, 20,
                     dtype=np.uint64)
```

N must be an **integer** for this to evaluate to `True`:

```
(n1 > 0) & (n1 < N)
```

Fixation probability as a function of N

```
fixations = [
    simulation(
        N,
        s,
        repetitions
    ) for N in Nrange
]
```

Fixation probability as a function of N

```python
fixations = np.array(fixations)
fixations
```

```
array([[False, False, ..., False, False],
       [False, True, ..., False, False],
       , ...,
       [False, False, ..., True, False],
       [False, False, ..., False, False]],
      dtype=bool)
```

Fixation probability as a function of N

```
fixations = np.array(fixations)
mean = fixations.mean(axis=1)
sem = fixations.std(
    axis=1,
    ddof=1
) / np.sqrt(repetitions)
```

Approximation

Motoo Kimura
1924-1994
Japan & USA

Kimura's equation:

$$\frac{e^{-2s} - 1}{e^{-2Ns} - 1}$$

```
def kimura(N, s):
    return np.expm1(-2 * s) /
           np.expm1(-2 * N * s)
```

`expm1(x)` is **e^{x}-1** with better precision for small values of **x**

`kimura` works on arrays out-of-the-box

```
%timeit [kimura(N=N, s=s)
                    for N in Nrange]
%timeit kimura(N=Nrange, s=s)
```

1 loop, best of 3: **752 ms** per loop

1000 loops, best of 3: **3.91 ms** per loop

X200 faster!

Numexpr

Fast evaluation of element-wise array expressions using a vector-based virtual machine

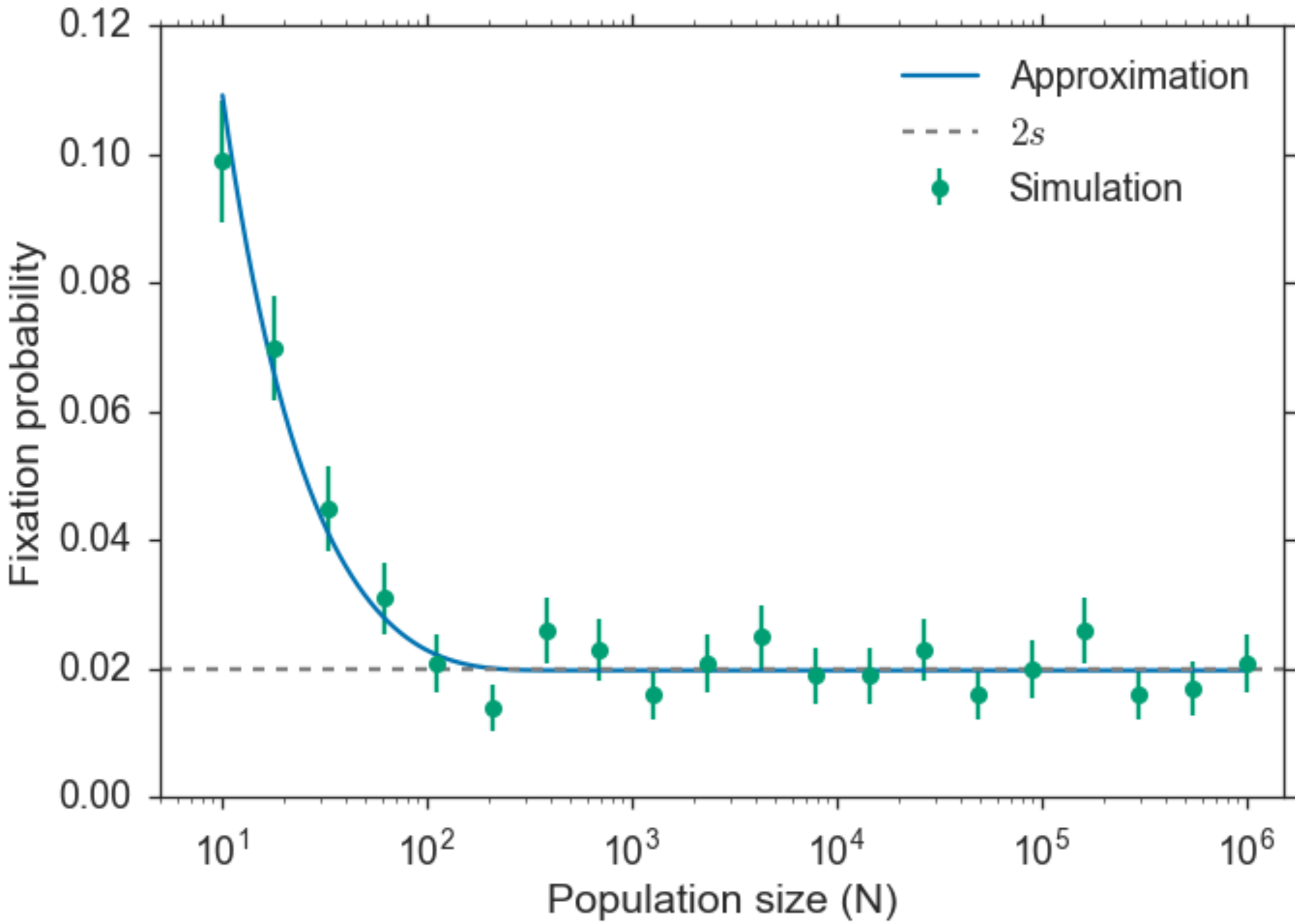
```
def kimura(N, s):
    return numexpr.evaluate(
        "expm1(-2 * s) /
         expm1(-2 * N * s)“)
```

```
%timeit kimura(N=Nrange, s=s)
```

1000 loops, best of 3: **803 µs** per loop **x5 faster**

https://github.com/pydata/numexpr

Plotting with matplotlib



http://matplotlib.org

Fixation time

How much time does it take variant **1** to go **extinct or to fix**?

We want to keep track of **time***.

time is measured in **number of generations**

https://commons.wikimedia.org/wiki/File:Prim_clockwork.jpg

```
def simulation(N, s, repetitions):
    n1 = np.ones(repetitions)
    T = np.empty_like(n1)
    update = (n1 > 0) & (n1 < N)
    t = 0

    while update.any():
        t += 1
        p = n1 * (1 + s) / (N + n1 * s)
        n1[update] = binomial(N, p[update])
        T[update] = t
        update = (n1 > 0) & (n1 < N)

    return n1 == N, T
```

`t` keeps track of time

```
def simulation(N, s, repetitions):
  n1 = np.ones(repetitions)
  T = np.empty_like(n1)
  update = (n1 > 0) & (n1 < N)
  t = 0

  while update.any():
    t += 1
    p = n1 * (1 + s) / (N + n1 * s)
    n1[update] = binomial(N, p[update])
    T[update] = t
    update = (n1 > 0) & (n1 < N)

  return n1 == N, T
```

T holds time for extinction/fixation

```
def simulation(N, s, repetitions):
  n1 = np.ones(repetitions)
  T = np.empty_like(n1)
  update = (n1 > 0) & (n1 < N)
  t = 0

  while update.any():
    t += 1
    p = n1 * (1 + s) / (N + n1 * s)
    n1[update] = binomial(N, p[update])
    T[update] = t
    update = (n1 > 0) & (n1 < N)

  return n1 == N, T
```

Return both Booleans and times (T)

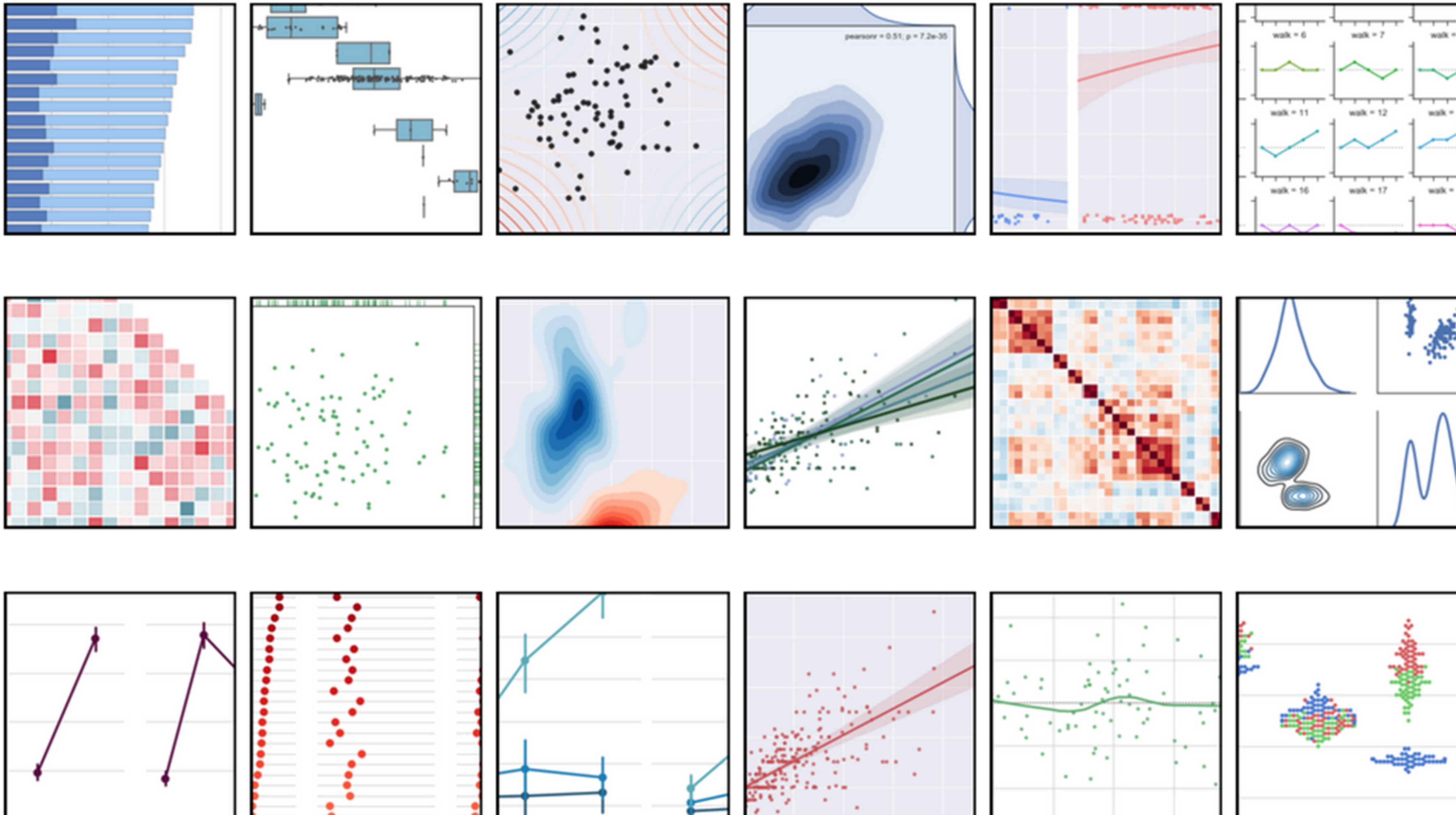
Statistical data visualization with Seaborn

- Visualization library based on **matplotlib** and **Pandas**
- High-level interface for attractive **statistical graphics**

By Michael Waskom, postdoc at NYU

http://seaborn.pydata.org

Statistical data visualization with Seaborn



Plot with Seaborn

```
from seaborn import distplot

fixations, times = simulation(…)

distplot(times[fixations])

distplot(times[~fixations])
```

Plot with Seaborn

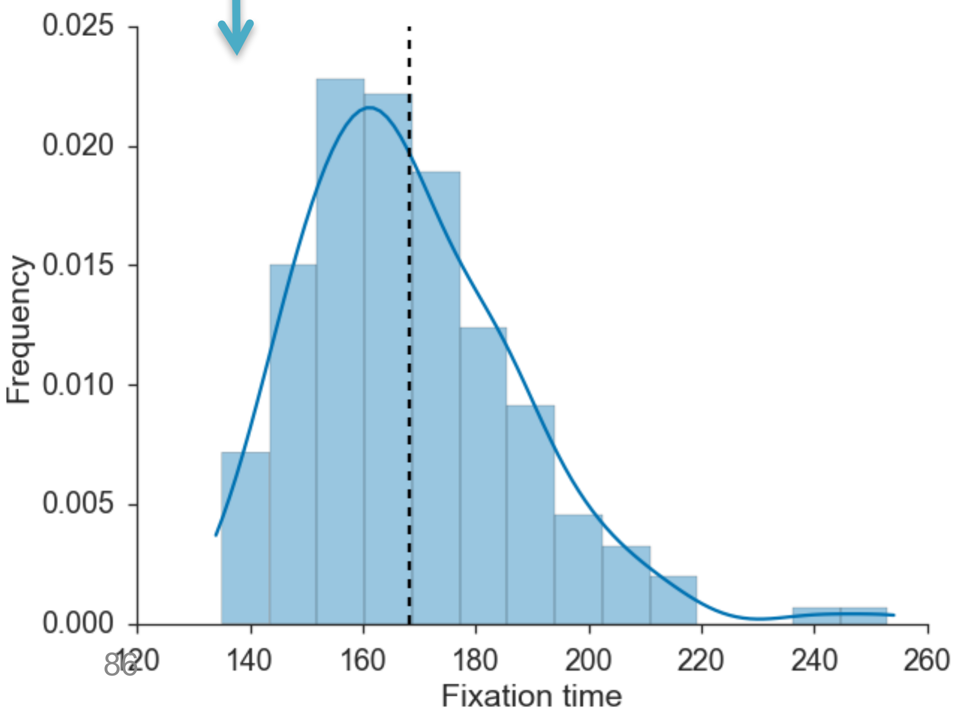
```
from seaborn import distplot

fixations, times = simulation(…)

distplot(times[fixations])

distplot(times[~fixations])
```

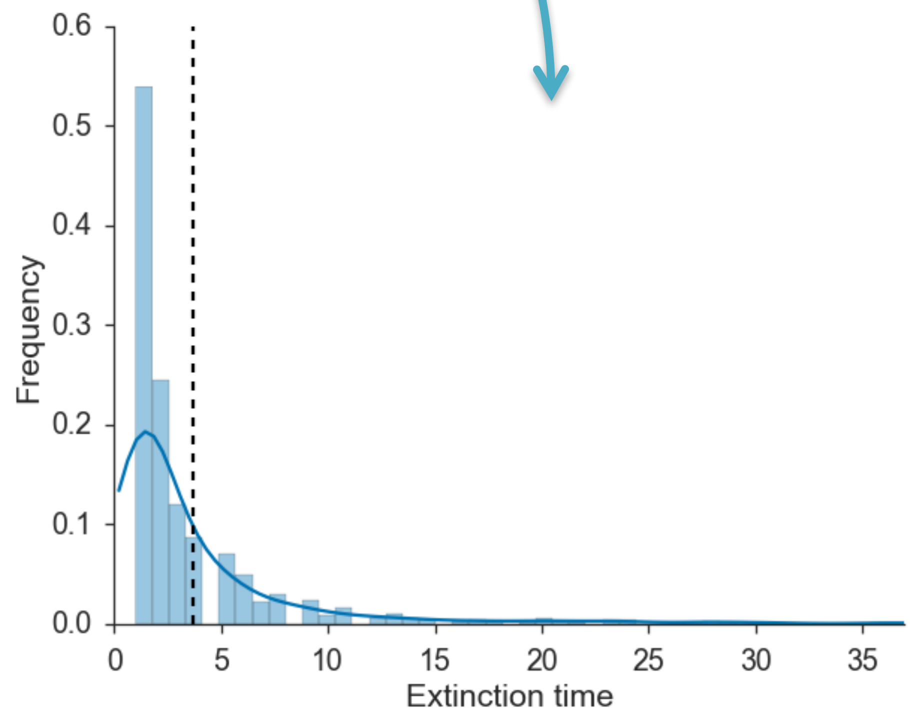
Plot with Seaborn

```
from seaborn import distplot

fixations, times = simulation(…)

distplot(times[fixations])

distplot(times[~fixations])
```

Plot with Seaborn

```
fixations, times = simulation(…)
distplot(times[fixations])
distplot(times[~fixations])
```





Diffusion equation approximation

$$I_1(x) = \frac{1 - e^{-2Nsx} - e^{-2Ns(1-x)} + e^{-2Ns}}{x(1-x)}$$

$$I_2(x) = \frac{(e^{2Nsx} - 1)(1 - e^{-2Ns(1-x)})}{x(1-x)}$$

$$J_1 = \frac{1}{s(1-e^{-2Ns})} \int_x^1 I_1(y)dy$$

$$J_2 = \frac{1}{s(1-e^{-2Ns})} \int_0^x I_2(y)dt$$

$$u = \frac{1 - e^{-2Nsx}}{1 - e^{-2Ns}}$$

$$\boxed{T_{fix} = J1 + \frac{1-u}{u} J_2}$$

Motoo Kimura
1924-1994
Japan & USA

Diffusion equation approximation

$$I_1(x) = \frac{1 - e^{-2Nsx} - e^{-2Ns(1-x)} + e^{-2Ns}}{x(1-x)}$$

$$I_2(x) = \frac{(e^{2Nsx} - 1)(1 - e^{-2Ns(1-x)})}{x(1-x)}$$

$$J_1 = \frac{1}{s(1 - e^{-2Ns})} \int_x^1 I_1(y)dy$$

$$J_2 = \frac{1}{s(1 - e^{-2Ns})} \int_0^x I_2(y)dt$$

$$u = \frac{1 - e^{-2Nsx}}{1 - e^{-2Ns}}$$

$$\boxed{T_{fix} = J1 + \frac{1-u}{u} J_2}$$

Requires integration...

Motoo Kimura
1924-1994
Japan & USA

```python
from functools import partial
from scipy.integrate import quad


def integral(f, N, s, a, b):
    f = partial(f, N, s)
    return quad(f, a, b)[0]
```

`integral` **will calculate** $\int_a^b f(N, s, x)dx$

```
from functools import partial
from scipy.integrate import quad


def integral(f, N, s, a, b):
  f = partial(f, N, s)
  return quad(f, a, b)[0]
```

`partial` **freezes** `N` **and** `s`
in `f(N, s, x)` **to create** `f(x)`

```python
from functools import partial
from scipy.integrate import quad


def integral(f, N, s, a, b):
  f = partial(f, N, s)
  return quad(f, a, b)[0]
```

SciPy's quad **computes a definite integral** $\int_a^b f(x)dx$

(using a technique from the Fortran library QUADPACK)

```
def I1(N, s, x):
    …

def I2(N, s, x):
    …
```

$$I_1(x) = \frac{1 - e^{-2Nsx} - e^{-2Ns(1-x)} + e^{-2Ns}}{x(1-x)}$$

$$I_2(x) = \frac{(e^{2Nsx} - 1)(1 - e^{-2Ns(1-x)})}{x(1-x)}$$

$$J_1 = \frac{1}{s(1-e^{-2Ns})} \int_x^1 I_1(y)dy$$

$$J_2 = \frac{1}{s(1-e^{-2Ns})} \int_0^x I_2(y)dt$$

$$u = \frac{1 - e^{-2Nsx}}{1 - e^{-2Ns}}$$

$$T_{fix} = J1 + \frac{1-u}{u} J_2$$

`I1` and `I2` are defined according to the equations

```python
@np.vectorize
def T_kimura(N, s):
  x = 1.0 / N
  J1 = -1.0 / (s * expm1(-2 * N * s)) *
          integral(I1, N, s, x, 1)
  J2 = -1.0 / (s * expm1(-2 * N *s)) *
          integral(I2, N, s, 0, x)
  u = expm1(-2 * N * s * x) /
      expm1(-2 * N * s)

  return J1 + ((1 - u) / u) * J2
```

`T_kimura` is the fixation time given a single copy of variant `1`: frequency `x=1/N`

```python
@np.vectorize
def T_kimura(N, s):
    x = 1.0 / N
    J1 = -1.0 / (s * expm1(-2 * N * s)) *
            integral(I1, N, s, x, 1)
    J2 = -1.0 / (s * expm1(-2 * N *s)) *
            integral(I2, N, s, 0, x)
    u = expm1(-2 * N * s * x) /
        expm1(-2 * N * s)

    return J1 + ((1 - u) / u) * J2
```

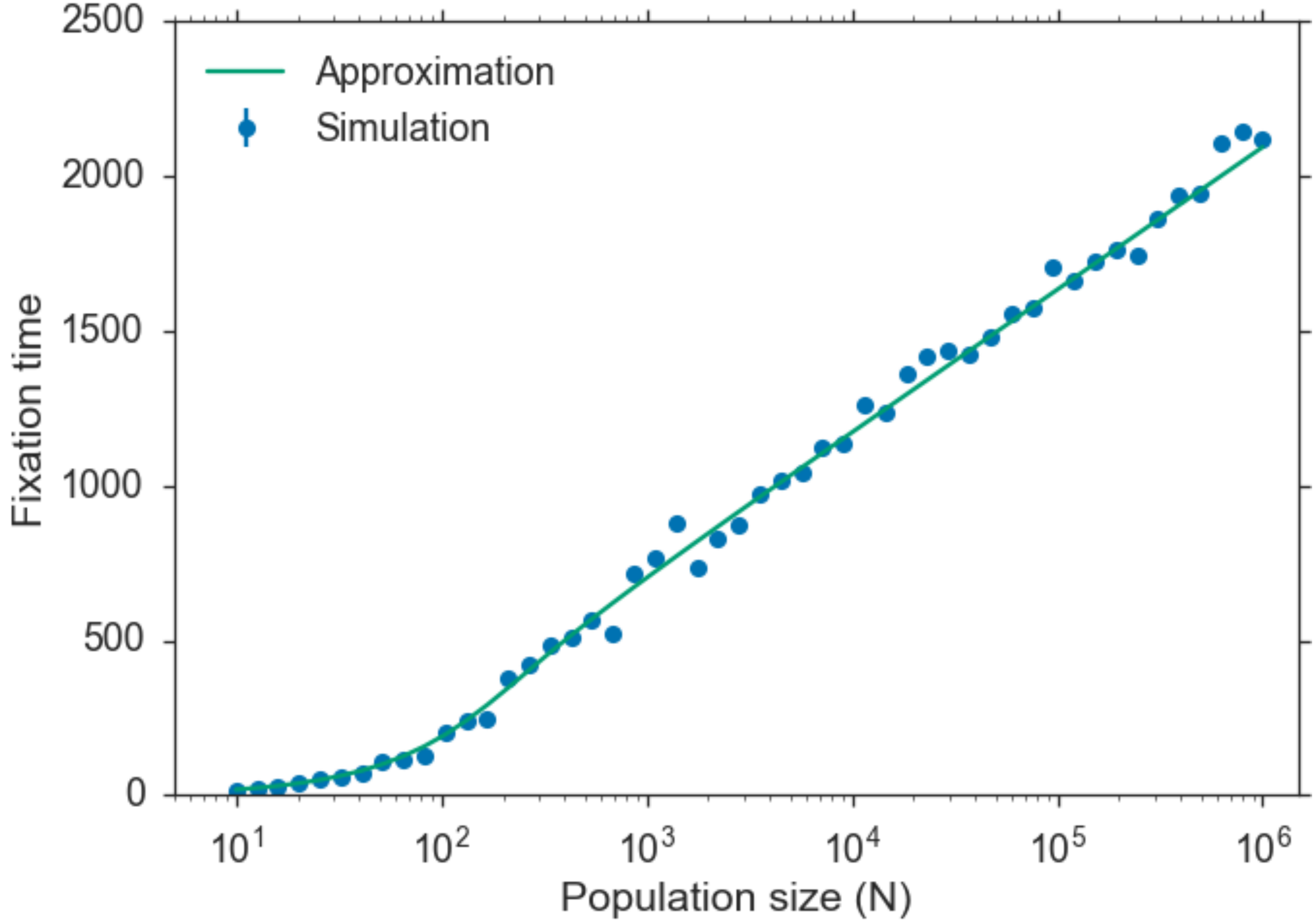
`J1` and `J2` are calculated using integrals of `I1` and `I2`

```python
@np.vectorize
def T_kimura(N, s):
    x = 1.0 / N
    J1 = -1.0 / (s * expm1(-2 * N * s)) *
            integral(I1, N, s, x, 1)
    J2 = -1.0 / (s * expm1(-2 * N *s)) *
            integral(I2, N, s, 0, x)
    u = expm1(-2 * N * s * x) /
        expm1(-2 * N * s)

    return J1 + ((1 - u) / u) * J2
```

T_{fix} is the return value

```python
@np.vectorize
def T_kimura(N, s):
    x = 1.0 / N
    J1 = -1.0 / (s * expm1(-2 * N * s)) *
             integral(I1, N, s, x, 1)
    J2 = -1.0 / (s * expm1(-2 * N *s)) *
             integral(I2, N, s, 0, x)
    u = expm1(-2 * N * s * x) /
        expm1(-2 * N * s)

    return J1 + ((1 - u) / u) * J2
```

`np.vectorize` **creates a function that takes a sequence and returns an array - x2 faster**


Approximation
Simulation
2500
2000
1500
1000
500
0
Fixation time
10^1
10^2
10^3
10^4
10^5
10^6
Population size (N)

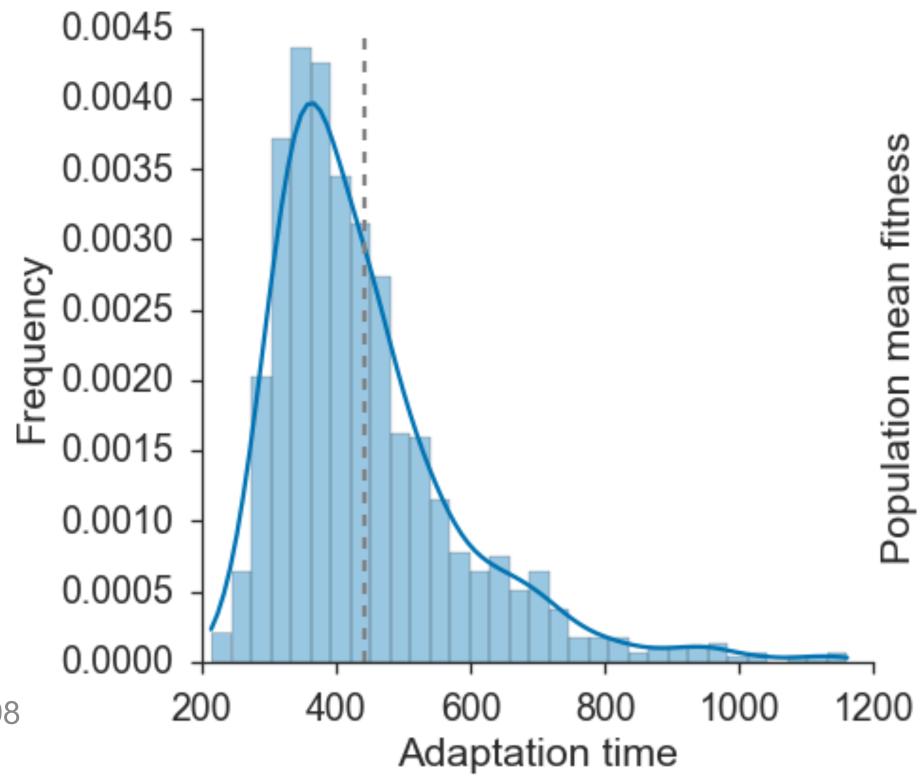
Dig Deeper

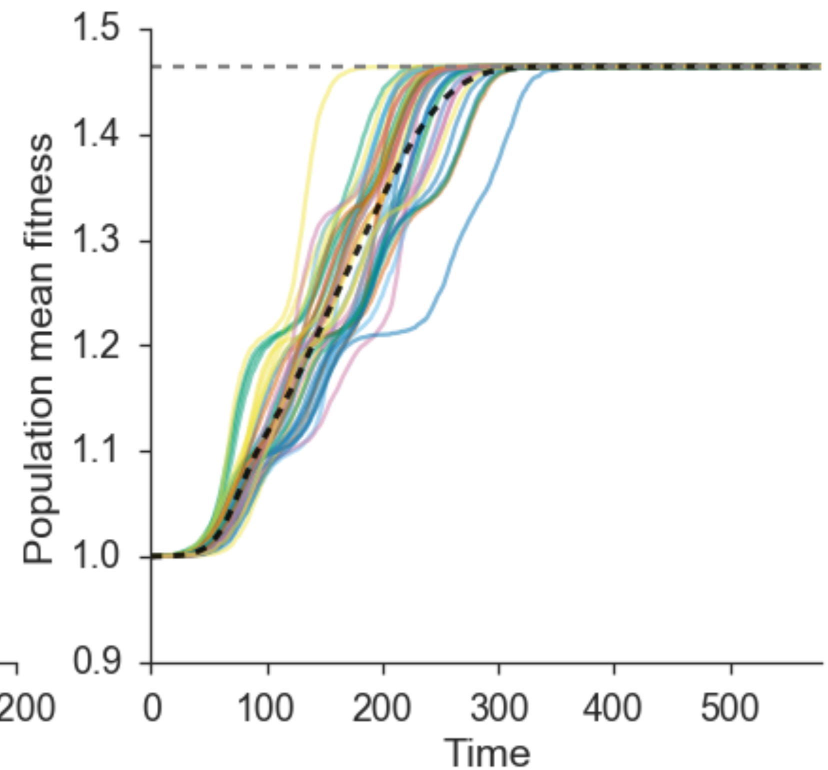
Online **Jupyter** notebook: github.com/yoavram/PyConIL2016

Multi-type simulation:

Includes **L** variants, with mutation.

Follow $\mathbf{n_0}$, $\mathbf{n_1}$, …, $\mathbf{n_L}$ until $\mathbf{n_L=N}$.





Dig Deeper

Online **Jupyter** notebook: github.com/yoavram/PyConIL2016

- Numba: JIT compiler, **array-oriented and math-heavy** Python syntax to machine code
- IPyParallel: IPython's sophisticated and powerful architecture for **parallel** and **distributed computing**.
- IPyWidgets: **Interactive HTML Widgets** for Jupyter notebooks and the IPython kernel

Thank You!

Presentation & Jupyter notebook : **git.io/14032017**

YOAV RAM
PYTHON TRAINING

yoav@yoavram.com

@yoavram

github.com/yoavram

www.yoavram.com

python.yoavram.com

Death to the Stock Photo