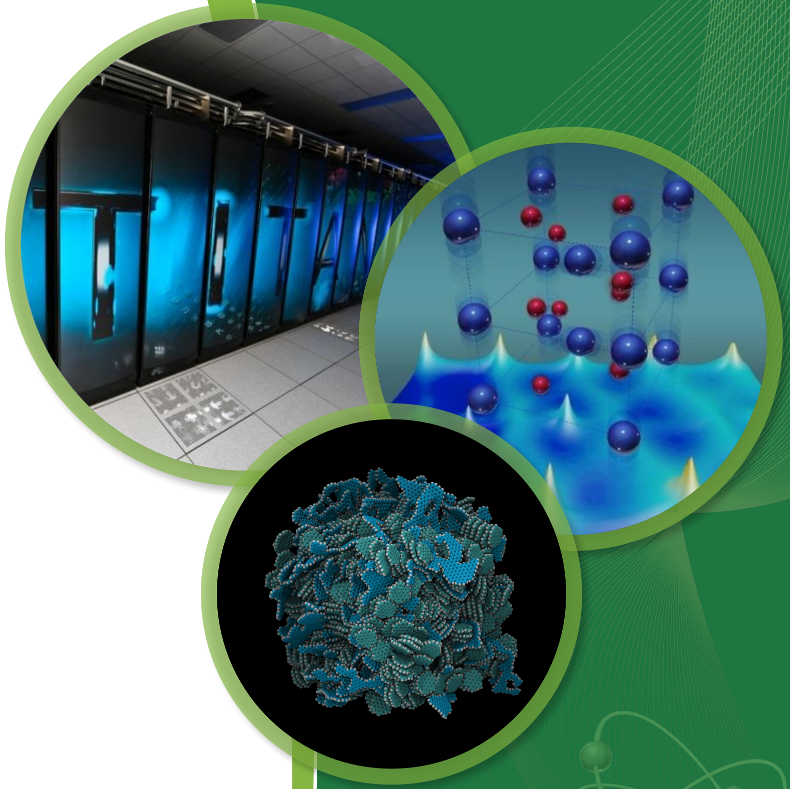
LEAPing Forward

A tour through an open-source evolutionary algorithm toolkit

Mark Coletti `<colettima@ornl.gov>`



LEAPing Forward

I will share aspects of ongoing work developing an open-source, python-based evolutionary algorithm (EA) toolkit.

I will cover details that will hopefully be useful to other python users.

What is an Evolutionary Algorithm?

- Biologically inspired
- Populations of individuals that represent posed solutions
- Selected parents beget offspring
- Offspring are incrementally different from their parents
  - via mutation
  - and possibly crossover
- “Survival of the fittest” culls inferior individuals
- Over time the population converges on viable solutions

(Public Domain image from https://www.flickr.com/photos/dennism2/)

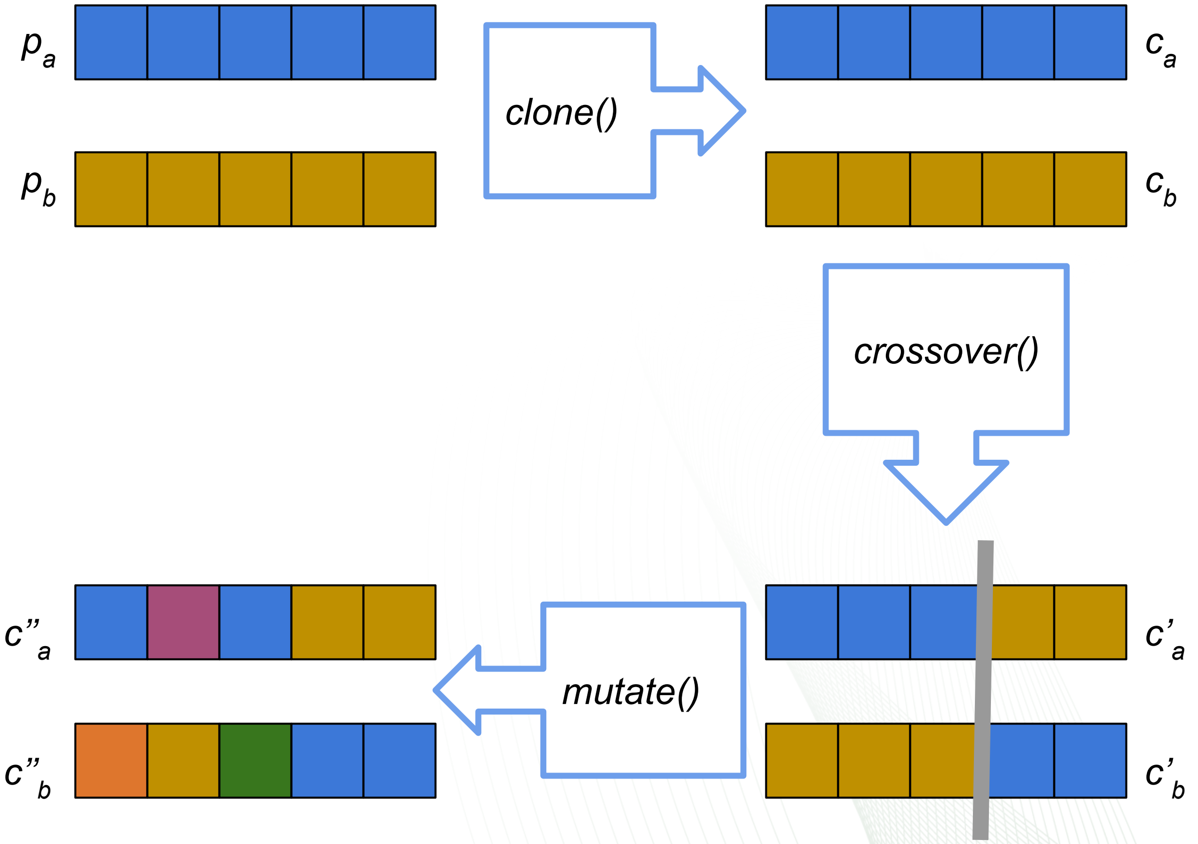
General Evolutionary Algorithm

main()

1. make initial population
2. evaluate them
3. current pop ← initial population
4. while not done:
   a. offspring ← *breed*(current pop)
   b. evaluate offspring
   c. current pop ← *cull*(current pop, offspring)

breed()

1. C <- {}
2. while *size*(C) < desired kids:
   a. $\{p_a,p_b\}$ ← *select*(P_i)
   b. $\{c_a,c_b\}$ ← *clone*(p_a,p_b)
   c. $\{c'_a,c'_b\}$ ← *crossover*(c_a,c_b)
   d. c''_a ← *mutate*(c'_a)
   e. c''_b ← *mutate*(c'_b)
   f. append $\{c''_a,c''_b\}$ to C
3. return C



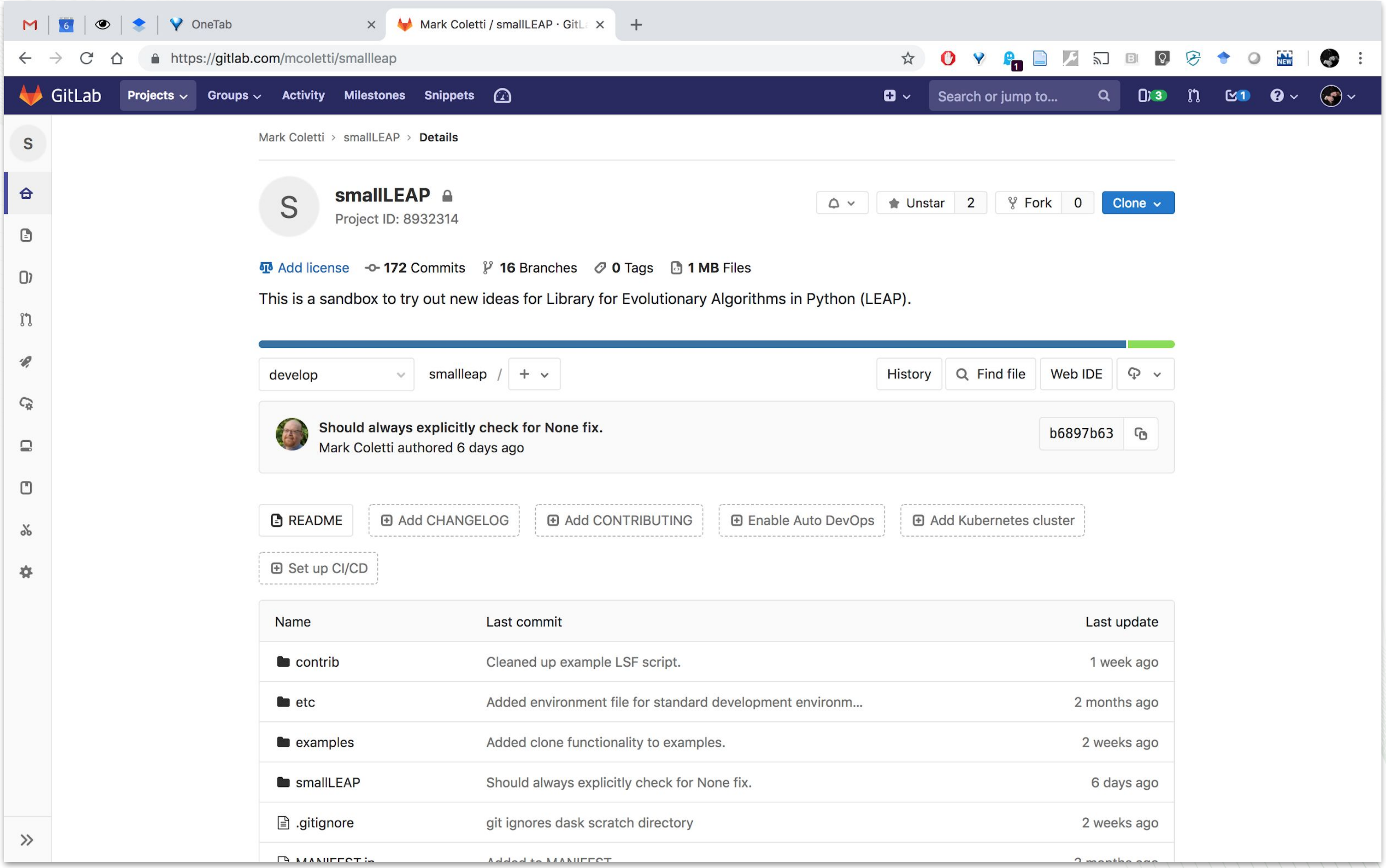
LEAP — a python-based open source EA toolkit

Ideally would like an available open-source EA solution.

- Library for Evolutionary Algorithms in Python (LEAP)
  - Thanks to Dr. Jeff Bassett of GMU
- Initially written in 2004 (!)
  - Python 2.4 was the most recent version that year
- Has been maintained, but suffers from a little bit-rot
- Had a unique pipeline-like architecture that was ripe for refactoring to take advantage of some modern python constructs
  - Closures
  - Generators
  - Functional programming

OAK RIDGE National Laboratory

smallLEAP as sandbox for LEAP refactoring



Jump right into the weeds

Let’s look at *some* of smallLEAPs implementation and design details.

Even though you may not have an interest in EAs, there may be some facets of smallLEAP that could be applicable to your python work.

Toolz is a useful 3rd party python package.

https://github.com/pytoolz/toolz

https://toolz.readthedocs.io/en/latest/

It provides three categories of tools:

- Tools for iterators
- Tools for functional programming
- Tools for dictionaries

If you’re familiar with the *nix style text processing pipeline:

```
grep newacronym acronyms.tex |  awk '{print $1}' | egrep -o '{[a-z]{4}}' | sed -e 's/{//' -e 's/}//'
alcc
dice
dsge
gdal
glcm
lulc
ccsi
```


OAK RIDGE National Laboratory

`toolz.functoolz.pipe()` serves a similar purpose

`toolz.functoolz.pipe(data, *funcs)`

- pipe data through a function sequence

```
>>> double = lambda i: 2 * i
>>> pipe(3, double, str)
'6'
```

(https://toolz.readthedocs.io/en/latest/api.html#toolz.functoolz.pipe)

From a certain perspective, an EA is a pipeline of data

1. The initial data is the set of prospective parents
2. Selected parents move down the pipe
3. They are cloned and passed down the pipe
4. Those clones are mutated and may go through crossover and passed down the pipe
5. The children are evaluated and passed down the pipe
6. Process is repeated until a set number of children are pooled, and then passed down the pipe
7. Survivors are culled to become prospective parents for the next generation

toolz.pipe() facilitates implementing an EA

```python
# This will loop until we've exhausted our budget of generations, or there
# has been no difference between successive sets of parents.
while all([max_gen(), not my_no_change(parent_population)]):
    new_parents = toolz.pipe(
        parent_population,
        smallLEAP.selection.tournament_select_generator, # return best of two randomly selected parents
        smallLEAP.reproduction.clone_generator, # clone them so we don't damage original parent
        functools.partial(smallLEAP.reproduction.bit_flip_mutation_generator, # binary bit flip mutation
                          probability=0.2),
        problem.evaluate_generator, # figure out fitness of new kid
        functools.partial(smallLEAP.reproduction.create_pool, size=20), # collect a pool of 20 offspring
        functools.partial(smallLEAP.selection.truncate, # pick only the best of parents *and* offspring
                          second_population=parent_population,
                          size=len(parent_population)))

    parent_population = new_parents
```


Oak Ridge National Laboratory

Using generators for selecting parents biased by fitness

```python
def tournament_selection(population, k=2):
    """ return best of drawn k samples

    defaults to "binary tournament" with k = 2

    Note that random.sample() samples *without replacement*, which means
    that selected individuals are guaranteed to be unique.  This may not
    be what you want; if so, consider random.choices(), instead.
    """
    choices = random.sample(population, k)

    return max(choices)


def tournament_select_generator(population, k = 2):
    """ Selects the best individual from k individuals randomly selected from
    the given population
    """
    while True:
        yield tournament_selection(population, k)
```


Oak Ridge National Laboratory

Generator to pass a clone down the pipeline

```python
def clone_generator(next_individual):
    """ generator version of individual.clone()

    :param next_individual: iterator for next individual to be cloned
    :return: copy of next_individual
    """
    while True:
        yield next(next_individual).clone()
```

More use of generators to implement mutation

```python
def binary_flip_mutation(individual, probability):
    """ return a copy of individual with with individual.sequence bits flipped
        based on probability
    """
    def flip(gene):
        if random.random() < probability:
            return (gene + 1) % 2
        else:
            return gene

    individual.encoding.sequence = [flip(gene) for gene in individual.encoding.sequence]

    return individual


def bit_flip_mutation_generator(next_individual, probability=0.1):
    """ Generator for mutating an individual and passing it down the line
    """
    while True:
        yield binary_flip_mutation(next(next_individual), probability)
```

Generator function for evaluating individuals as needed

```python
def evaluate_generator(next_individual):
    """ Evaluates the next individual in the pipeline and passes them along.
    """
    while True:
        individual = next(next_individual)
        individual.evaluate()
        yield individual
```


OAK RIDGE
National Laboratory

create_pool() pulls offspring from upstream as needed

```python
def create_pool(next_individual, size):
    """ 'Sink' for creating `size` individuals from preceding pipeline source.

    Allows for "pooling" individuals to be processed by next pipeline
    operator.  Typically used to collect offspring from preceding set of
    selection and birth operators, but could also be used to, say, "pool"
    individuals to be passed to an EDA as a training set.
    """
    return [next(next_individual) for _ in range(size)]
```

toolz.itertoolz.topk() and itertools.chain()

```python
def truncate(population, size, second_population=None):
    """ return the `size` best individuals from the given population

        second_population is an optional population that is "blended" with the
        first for truncation purposes, and is usually used to allow parents
        and offspring to compete.
    """
    if second_population is not None:
        return toolz.itertoolz.topk(size, itertools.chain(population,
                                                          second_population))
    else:
        return toolz.itertoolz.topk(size, population)
```

Using function closures to impose max generations

```python
def countdown(limit):
    """ Used to countdown to zero.

        Repeated invocations will return True until limit reached, then
        False is returned. Used for determining maximum generations to run.

        countdown.count() returns current count
    """
    curr = limit

    def count():
        return curr

    def do_countdown():
        nonlocal curr
        curr -= 1
        if curr < 0:
            return False
        else:
            return True

    do_countdown.count = count

    return do_countdown
```

Expanded example

```python
# run for a maximum of 8 generations
max_gen = smallLEAP.halt.countdown(8)

# run until two successive generations produce identical parents
my_no_change = smallLEAP.halt.no_change()

# write offspring to this CSV file
my_csv_probe = smallLEAP.probes.population_probe('offspring.csv')

# This will loop until we've exhausted our budget of generations, or there
# has been no difference between successive sets of parents.
while all([max_gen(), not my_no_change(parent_population)]):
    new_parents = toolz.pipe(
        parent_population,
        smallLEAP.selection.tournament_select_generator,
        smallLEAP.reproduction.clone_generator,
        functools.partial(smallLEAP.reproduction.uniform_recombination_generator
                          p_swap=0.5),
        functools.partial(smallLEAP.reproduction.bit_flip_mutation_generator,
                          probability=0.2),
        problem.evaluate_generator,
        functools.partial(smallLEAP.reproduction.create_pool, size=20),
        my_csv_probe,  # snapshot of offspring before survival op
        functools.partial(smallLEAP.selection.truncate,
                          second_population=parent_population,
                          size=len(parent_population)))

    print('new parents:', [_ for _ in new_parents])

    parent_population = new_parents
```

Highlights

- Exposed to functional- and iterator-oriented packages
  - functools — https://docs.python.org/3.6/library/functools.html
  - itertools — https://docs.python.org/3.6/library/itertools.html
  - toolz — https://github.com/pytoolz/toolz
- Can use iterators to de-couple functionality for enhanced flexibility
- Usefulness of closures and generators
- SmallLEAP is on gitlab
  - Private repo, ask to be brought in after creating a gitlab.com account
  - https://gitlab.com/mcoletti/smallleap
  - Eventually will become LEAP 2.0

Thank you.

Mark Coletti <colettima@ornl.gov>

This presentation is at: https://goo.gl/Z6312P