Welcome!

WRITING EFFICIENT PYTHON CODE

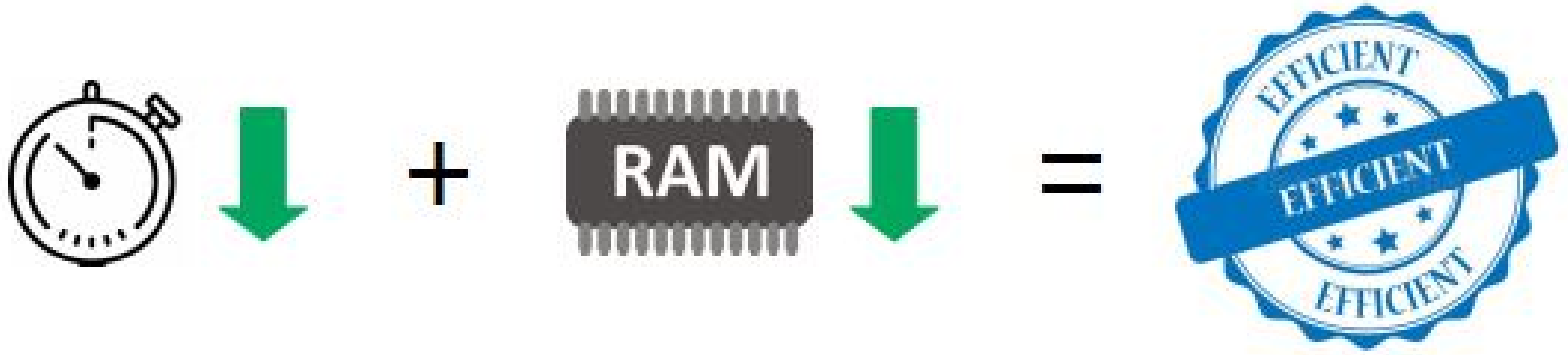
Logan Thomas
Scientific Software Technical Trainer, Enthought

Course overview

- Your code should be a tool used to gain insights
  - Not something that leaves you waiting for results
- In this course, you will learn:
  - How to write clean, fast, and efficient Python code
  - How to profile your code for bottlenecks
  - How to eliminate bottlenecks and bad design patterns

Defining efficient

- Writing ***efficient*** Python code
  - Minimal completion time (*fast runtime*)
  - Minimal resource consumption (*small memory footprint*)



Defining Pythonic

- Writing efficient ***Python*** code
  - Focus on readability
  - Using Python's constructs as intended (i.e., *Pythonic*)

```python
# Non-Pythonic
doubled_numbers = []

for i in range(len(numbers)):
    doubled_numbers.append(numbers[i] * 2)
# Pythonic
doubled_numbers = [x * 2 for x in numbers]
```

The Zen of Python by Tim Peters

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
...
```

Things you should know

- Data types typically used in Data Science
  - Data Types for Data Science
- Writing and using your own functions
  - Python Data Science Toolbox (Part 1)
- Anonymous functions (`lambda` expressions)
  - Python Data Science Toolbox (Part 1)
- Writing and using list comprehensions
  - Python Data Science Toolbox (Part 2)

Let's get started!

WRITING EFFICIENT PYTHON CODE

Building with built-ins

WRITING EFFICIENT PYTHON CODE

Logan Thomas
Scientific Software Technical Trainer, Enthought

datacamp

The Python Standard Library

- Python 3.6 Standard Library
  - Part of every standard Python installation
- Built-in types
  - `list` , `tuple` , `set` , `dict` , and others
- Built-in functions
  - `print()` , `len()` , `range()` , `round()` , `enumerate()` , `map()` , `zip()` , and others
- Built-in modules
  - `os` , `sys` , `itertools` , `collections` , `math` , and others

Built-in function: range()

Explicitly typing a list of numbers

```
nums = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Using `range()` to create the same list

```python
# range(start,stop)
nums = range(0,11)

nums_list = list(nums)
print(nums_list)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```python
# range(stop)
nums = range(11)

nums_list = list(nums)
print(nums_list)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Built-in function: range()

Using `range()` with a step value

```python
even_nums = range(2, 11, 2)


even_nums_list = list(even_nums)
print(even_nums_list)
```

```
[2, 4, 6, 8, 10]
```

Built-in function: enumerate()

Creates an indexed list of objects

```
letters = ['a', 'b', 'c', 'd' ]

indexed_letters = enumerate(letters)

indexed_letters_list = list(indexed_letters)
print(indexed_letters_list)
```

```
[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd')]
```

Built-in function: enumerate()

Can specify a start value

```
letters = ['a', 'b', 'c', 'd' ]

indexed_letters2 = enumerate(letters, start=5)

indexed_letters2_list = list(indexed_letters2)
print(indexed_letters2_list)
```

```
[(5, 'a'), (6, 'b'), (7, 'c'), (8, 'd')]
```

Built-in function: map()

Applies a function over an object

```
nums = [1.5, 2.3, 3.4, 4.6, 5.0]

rnd_nums = map(round, nums)

print(list(rnd_nums))
```

```
[2, 2, 3, 5, 5]
```

Built-in function: map()

`map()` with `lambda` (anonymous function)

```
nums = [1, 2, 3, 4, 5]

sqrd_nums = map(lambda x: x ** 2, nums)

print(list(sqrd_nums))
```

```
[1, 4, 9, 16, 25]
```

Let's start building with built-ins!

WRITING EFFICIENT PYTHON CODE

The power of NumPy arrays

WRITING EFFICIENT PYTHON CODE

Logan Thomas
Scientific Software Technical Trainer, Enthought

NumPy array overview

- Alternative to Python lists

```
nums_list = list(range(5))
```

```
[0, 1, 2, 3, 4]
```

```
import numpy as np

nums_np = np.array(range(5))
```

```
array([0, 1, 2, 3, 4])
```

NumPy arrays vectorize operations, so they are performed on all elements of an object at once. This allows us to efficiently perform calculations over entire arrays. Notice that by squaring the array nums_np, all elements are squared at once.

```
# NumPy array homogeneity
nums_np_ints = np.array([1, 2, 3])
```

```
array([1, 2, 3])
```

```
nums_np_ints.dtype
```

```
dtype('int64')
```

```
nums_np_floats = np.array([1, 2.5, 3])
```

```
array([1. , 2.5, 3. ])
```

```
nums_np_floats.dtype
```

```
dtype('float64')
```

NumPy array broadcasting

- Python lists don't support broadcasting

```
nums = [-2, -1, 0, 1, 2]
nums ** 2
```

```
TypeError: unsupported operand type(s) for ** or pow(): 'list' and 'int'
```

- List approach

```
# For loop (inefficient option)
sqrd_nums = []
for num in nums:
    sqrd_nums.append(num ** 2)
print(sqrd_nums)
```

```
[4, 1, 0, 1, 4]
```

```
# List comprehension (better option but not best)
sqrd_nums  = [num ** 2 for num in nums]


print(sqrd_nums)
```

```
[4, 1, 0, 1, 4]
```

NumPy array broadcasting

- NumPy array broadcasting for the win!

```
nums_np = np.array([-2, -1, 0, 1, 2])
nums_np ** 2
```

```
array([4, 1, 0, 1, 4])
```

Basic 1-D indexing (lists)

```
nums = [-2, -1, 0, 1, 2]
nums[2]
```

```
0
```

```
nums[-1]
```

```
2
```

```
nums[1:4]
```

```
[-1, 0, 1]
```

Basic 1-D indexing (arrays)

```
nums_np = np.array(nums)
nums_np[2]
```

```
0
```

```
nums_np[-1]
```

```
2
```

```
nums_np[1:4]
```

```
array([-1, 0, 1])
```

```
# 2-D list
nums2 = [ [1, 2, 3],
          [4, 5, 6] ]
```

```
# 2-D array

nums2_np = np.array(nums2)
```

- Basic 2-D indexing (lists)

```
nums2[0][1]
```

```
2
```

```
[row[0] for row in nums2]
```

```
[1, 4]
```

- Basic 2-D indexing (arrays)

```
nums2_np[0,1]
```

```
2
```

```
nums2_np[:,0]
```

```
array([1, 4])
```

8. NumPy array indexing
When using two-dimensional arrays and lists, the advantages of arrays are clear. To return the second item of the first row in our two-dimensional object, the array syntax is square bracket, zero, comma, one, square bracket. The analogous list syntax is a bit more verbose as you have to surround both the zero and one with square brackets. To return the first column of values in our two-d object, the array syntax is square bracket, colon, comma, zero, square bracket. Lists don't support this type of syntax, so we must use a list comprehension to return columns.

NumPy array boolean indexing

```
nums = [-2, -1, 0, 1, 2]
nums_np =  np.array(nums)
```

9. NumPy array boolean indexing
NumPy arrays also have a special technique called boolean indexing. Suppose we wanted to gather only positive numbers from the sequence listed here. With an array, we can create a boolean mask using a simple inequality. Indexing the array is as simple as enclosing this inequality in square brackets.

- Boolean indexing

```
nums_np > 0
```

```
array([False, False, False,  True,  True])
```

```
nums_np[nums_np > 0]
```

```
array([1, 2])
```

- No boolean indexing for lists

```
# For loop (inefficient option)
pos = []
for num in nums:
    if num > 0:
        pos.append(num)
print(pos)
```

```
[1, 2]
```

```
# List comprehension (better option but not best)
pos = [num for num in nums if num > 0]
print(pos)
```

```
[1, 2]
```

Let's practice with powerful NumPy arrays!

WRITING EFFICIENT PYTHON CODE