# Midterm info, Generators, Lambdas

COMP3522 OBJECT ORIENTED PROGRAMMING 2

WEEK 6

## MIDTERM INFO

## Midterm info

- Feb 18th 8:30am-10:00am. SW09-110
- Come 10 minutes early to make sure we don’t start late
  - I won’t let you in early, but be ready outside
- Paper exam. Closed book
- Sign out with me before leaving

- Covers everything from week 1 up to and including week 5. (Nothing from this week)
- If it’s in the slides and links in slides, it’s possibly on the exam

## Midterm format

- Multiple choice, True/False
- Read code snippet and write the output
- Ask to explain some concept, draw some diagram
- Short coding question
- Long coding question
    - Long questions are guided
    - Broken down into multiple parts, ask you to write individual functions

## STUDY STRATEGY

## Strategy

- Go over slides and links in slides
- Go over old quizzes
    - Activities > Quizzes > Quiz 2 (down arrow) > Submissions > Attempt 1
- Go over code samples I reference in slides sample.py
- Recall what we did in labs

## Strategy

- Something unclear in slides?
  - Search geeksforgeeks, tutorialspoint, programiz + topic
  - Ask me

- Think about topics we spent a lot of time on in lecture
  - If I went over something in depth, chances are I’m going to cover it

## What you DON’T need to know

- I won’t ask you to write code and also include the built in module
- Unless I explicitly ask you “which module do you need to include for X function to work?”
- Don’t worry about commenting code (*unless explicitly stated):

## Recap

- Iterators, Iterables and Iteration
- List Comprehensions
- Dictionary Comprehensions
- Functions as Objects

And so far.. Observer Pattern.

COMP 3522

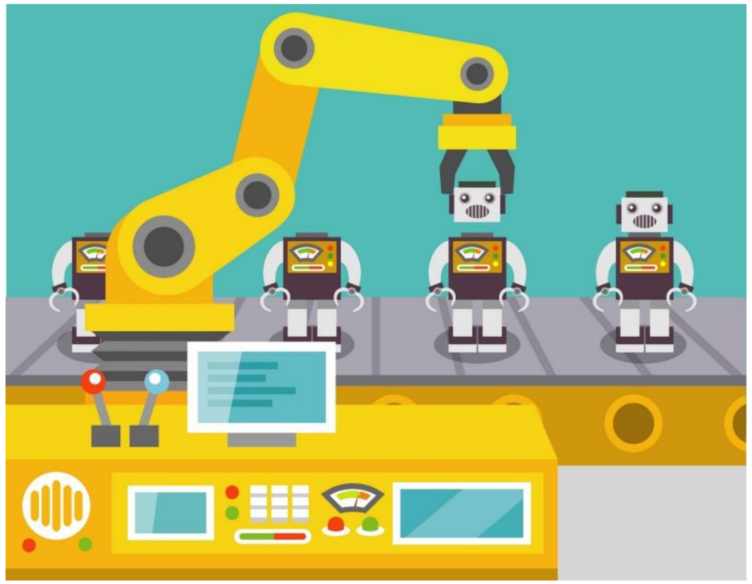
## Generators

## Generators

Generators are a special kind of iterator.

They operate over a sequence of objects / iterables.

Generators use a `yield` statement to return the next element in a sequence.



## Generators

So for a moment, let’s forget about generators and yield statements. Let’s look at ranges instead.

Ranges are awesome because:

- They save memory
- They are easy to read
- They provide all the advantages of being an iterable.

## Generators

```
my_range = range(1000)
a_list = [i for i in range(1000)]

print(my_range) #range(0,1000)
print(a_list) #[0,1,2, . . ., 997,998,999]

print(f"Range Size in bytes: {sys.getsizeof(my_range)}") #Range Size in
bytes: 24
print(f"List Size in bytes: {sys.getsizeof(a_list)}") #List Size in
bytes: 4508
```

why_generator.py

## Generators

Ranges aren’t exactly like generators but the concept is the same

Generators are special constructs and a special type of iterator that iterates over a sequence while saving memory.

Think of generators as objects that iterate over a sequence.

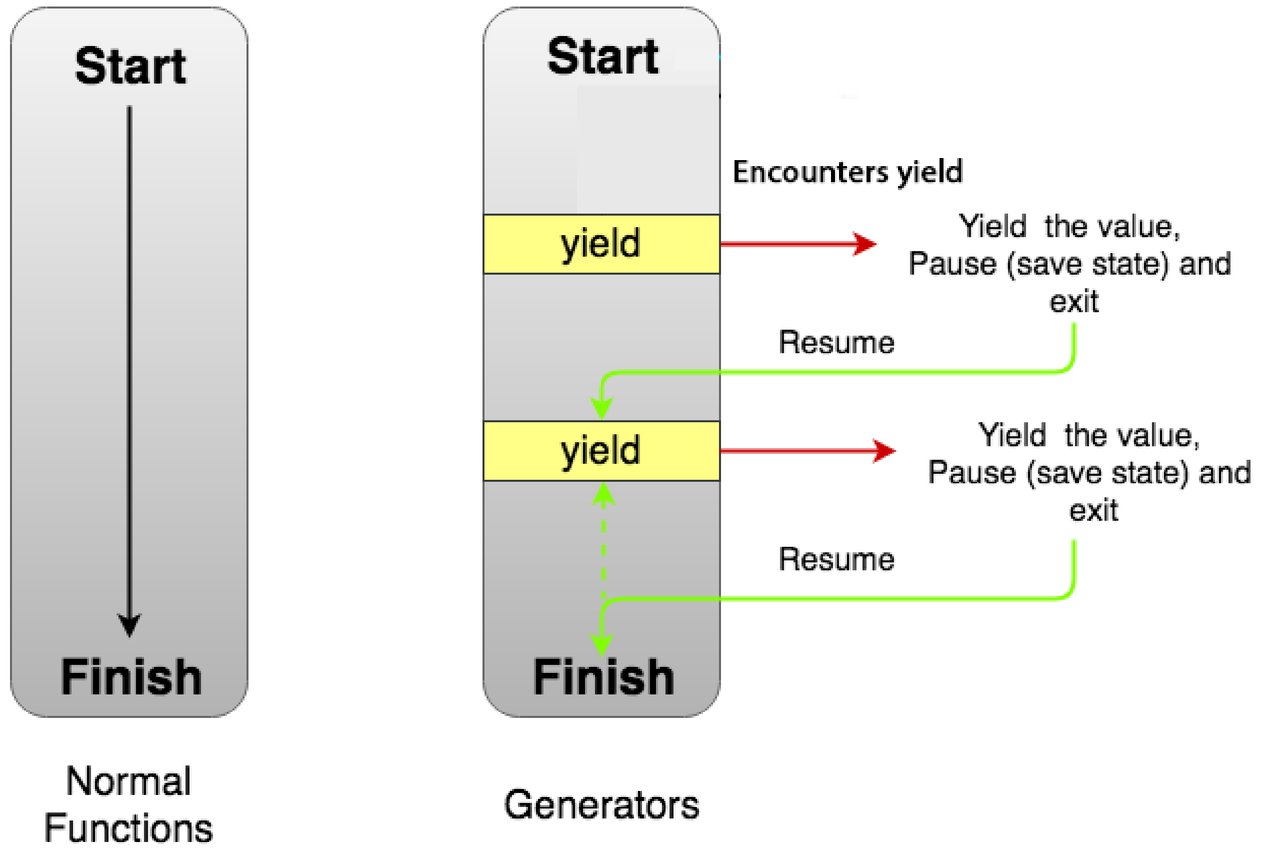
However instead of creating and storing the entire sequence at once, they only contain the code to *generate* the next object in a sequence

## Generators

How do we create a generator? Just have a yield statement in a function

As soon as a function has one or more yield statements, the function is now considered a generator

```
def my_function():
    n = 0
    #some code
    yield n #returns n and pauses. Execution continues from next line later
    #some more code
    yield n
```


Start
Finish
Normal Functions
Start
Encounters yield
yield
Yield the value, Pause (save state) and exit
Resume
yield
Yield the value, Pause (save state) and exit
Resume
Finish
Generators

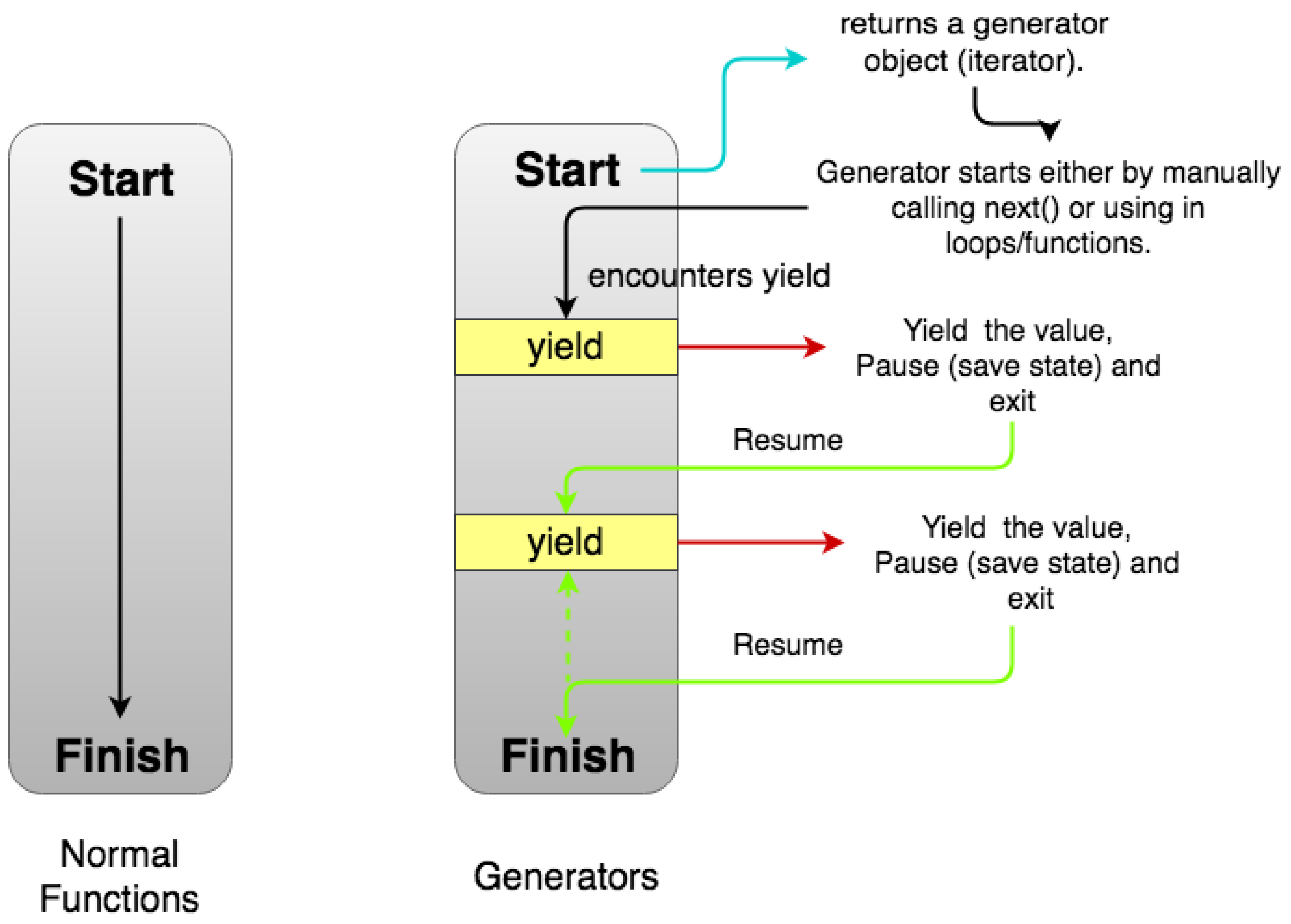
# Generators - Yield

The `yield` statement is a special statement.

Think of it like a return statement, but instead of returning from a method it returns the value and remembers its place in the method.

The next time the method is called, it resumes execution by executing the next line after the yield statement.

You are temporarily *'yielding'* the flow of control back to the entity that called the method.



returns a generator object (iterator).
Generator starts either by manually calling next() or using in loops/functions.
Start
Finish
Start
encounters yield
yield
Yield the value, Pause (save state) and exit
Resume
yield
Yield the value, Pause (save state) and exit
Resume
Finish
Normal Functions
Generators

# Generator Methods

When python sees a generator method with a yield statement it **wraps it up in an object**.

This object functions like an iterator.

Every time `next(generator_object)` is called, the generator method executes. This happens behind-the-scenes in

Subsequent calls to next continue from the instruction after the last yield statement.

generator_example.py

## Generators

Let’s go back to our log file example from last week.

How do we filter out all the lines with the [Warning] tag in it without creating a temporary variable that takes up memory?

```
user@host> file show /var/log/processes Feb 22 08:58:24 router1 snmpd[359]:
%DAEMON-3-SNMPD_TRAP_WARM_START: trap_generate_warm: SNMP trap: [Warning]
warm start
Feb 22 20:35:07 router1 snmpd[359]:
%DAEMON-6-SNMPD_THROTTLE_QUEUE_DRAINED: [Warning]
trap_throttle_timer_handler: cleared all throttled traps
Feb 23 07:34:56 router1 snmpd[359]:
%DAEMON-3-SNMPD_TRAP_WARM_START: trap_generate_warm: SNMP trap: warm start
Feb 23 07:38:19 router1 snmpd[359]:
%DAEMON-2-SNMPD_TRAP_COLD_START: trap_generate_cold: [Warning] SNMP trap:
cold start
```

concrete_generator.py

## Fibonacci sequence with generators

In mathematics, the **Fibonacci numbers**, commonly denoted $F_n$, form a sequence, called the **Fibonacci sequence**, such that each number is the sum of the two preceding ones, starting from 0 and 1.

0, 1

0, 1, 1

0, 1, 1, 2, 3

0, 1, 1, 2, 3, 5, 8

0, 1, 1, 2, 3, 5, 8, 13

0, 1, 1, 2, 3, 5, 8, 13, 21

0, 1, 1, 2, 3, 5, 8, 13 , 21, 36…

https://en.wikipedia.org/wiki/Fibonacci_number

## Fibonacci sequence with generators

Imagine creating a function that returns the next number in the Fibonacci sequence every time it's called

This might require some static or global variables to keep track of the numbers.

We might instead write the code in a class

- Classes can keep track of the numbers with instance variables
- Can write a get_next_fib() method to get the next number

Or we can use generators

## Fibonacci sequence with generators

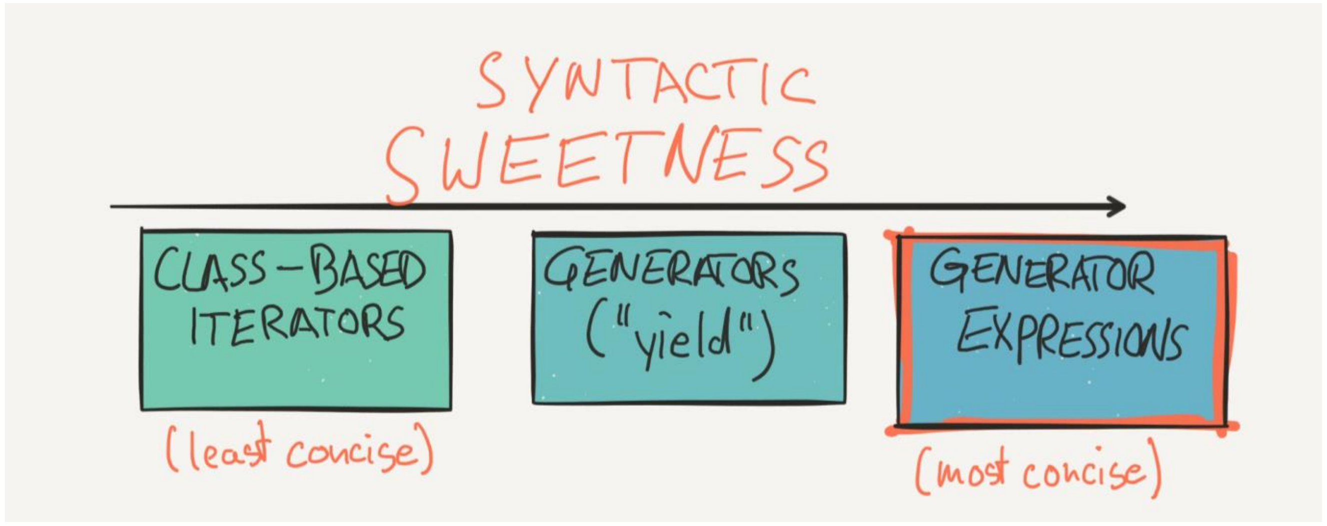
Infinite Fibonacci generator. Calling next(fib) will give the next number in Fibonacci sequence indefinitely

```
def fibonacci_sequence():
    current = 0
    previous = 1
    while True:
        yield current
        previous, current = current, previous + current

#call generator with for loop
fib = fibonacci_sequence()
for x in range(0,100):
    print(next(fib), end=", ")

Output:
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 … 7778742049]
```

## Generator Expressions

What if I told you that there was **an even better** way of using generators!

What if I told you that you could add even **more syntactic sugar** to your code!



## Generator Expressions

Similar to a list comprehension.

Syntactically it is exactly the same with the exception that we now use Parentheses – () instead of Square brackets – []

These are not as flexible as generator methods, but are very ***Clean*** .

- Not as flexible because these generators can not be re-used

```
( f(x) for x in some_iterable if condition )
```

# Generator Expressions

Say we want to
- Access all the even numbers from 0 – 100
- Multiply it by 5

By using a generator instead of a list comprehension we save memory.

A range uses generators in its implementation. But all the rules of a generator don’t apply to ranges.

The **ok** way of doing this:

```
gen_exp = ( x * 5 for x in range(100) if x % 2 == 0)
for i in gen_exp:
    print(i)
```

## Generator Expressions

```
gen_exp = ( x * 5 for x in range(100) if x % 2 == 0)

for i in gen_exp:

    print(i)
```

The code above is only ‘ok’ because it is not re-usable

Once we iterate over gen_exp, iterating over it again will produce no output

## Generator Expressions

The **good** way of doing this, would be to use the **`yield from`** syntax:

```
def gen_even_multiples(multiplier, start, stop):
        yield from (x * multiplier for x in range(start, stop) if x % 2 == 0)


for result in gen_even_multiples(5, 0, 100):

    print(result)
```

Notice how we wrapped the *generator expression* in a generator function

Keep in mind if we're calling generators within generators, use the yield from keyword to maintain the yield functionality

## Generator Expressions

Remember our previous log file parser? A function with multiple lines

```
def warning_filter_generator(file_object):
    for line in file_object:
        if '[Warning]' in line:
            yield line
```

```
with open("log_file.txt", mode='r', encoding='utf-8') as log_file:
    filter = warning_filter_generator(log_file)
    for warning_line in filter:
        print(warning_line)
```

generator_examples.py

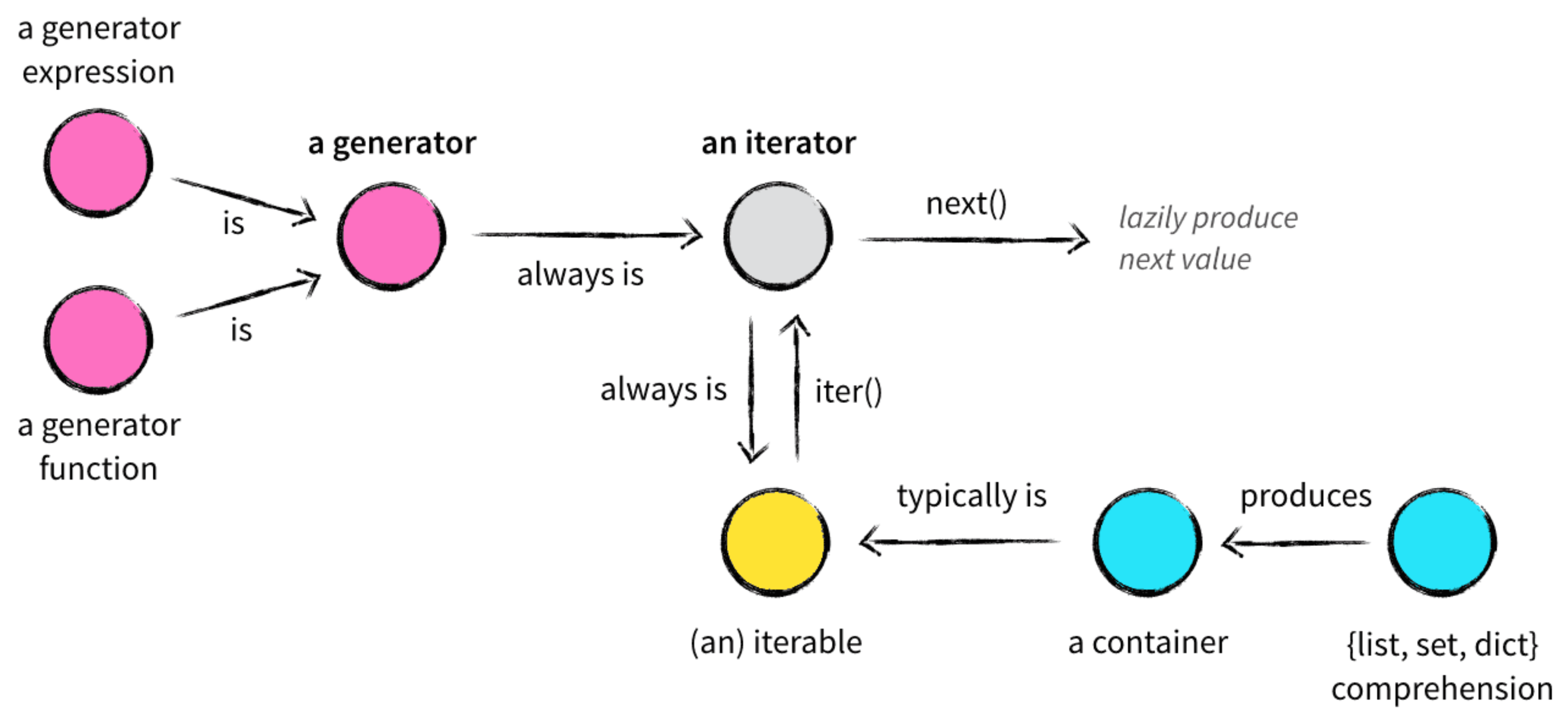
## Generator Expressions

We can make our previous log file parser even better now that we know about *generator expressions*:

```
def warning_filter_generator(file_object):
    #one line!
    yield from (line for line in file_object if '[Warning]' in line)


with open("log_file.txt", mode='r', encoding='utf-8') as log_file:
    filter = warning_filter_generator(log_file)
    for warning_line in filter:
        print(warning_line)
```

generator_examples.py

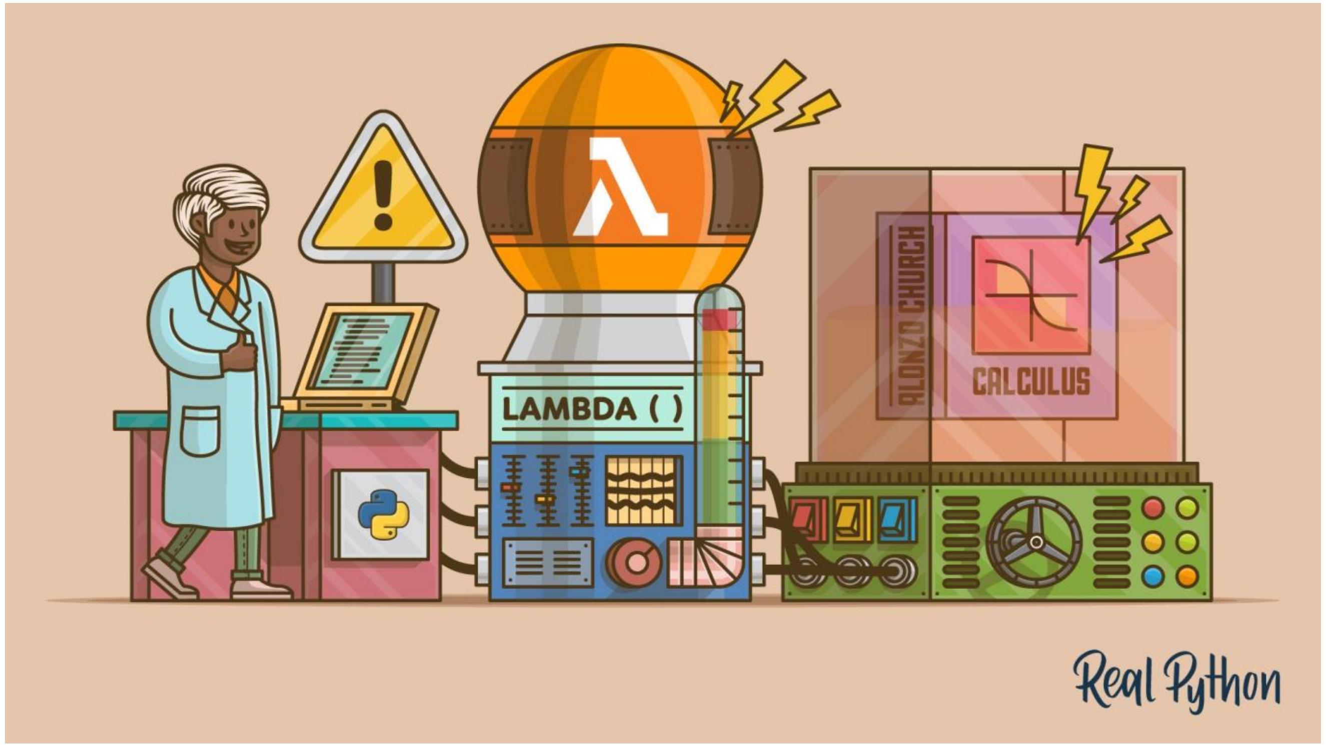
# Generators - Summary



## Generators - Summary

- Generators are a special kind of an iterator
- They make use of yield statements
- They save memory by not actually reserving a contiguous chunk of memory for an iterable.
- They should be used whenever you need to work on an iterable an element at a time, but don’t want to actually change the iterable itself or store the result in memory
- Extremely useful when working with complex objects and Big Data. (Who knew big data takes up a lot of memory?)

# Lambda Functions



# Lambda Functions – Anonymous Functins

Lambda Functions are also known as Anonymous Functions

These are functions without a name

Think of defining a temporary function which has a small scope.

Syntax:

```
lambda arguments: expression
```

Regular function:

```
def my_func(arguments):
    expression
```

## Lambda Functions – Anonymous Functins

Lambda Functions are also known as Anonymous Functions

These are functions without a name

Think of defining a temporary function which has a small scope.

Syntax:

```
lambda arguments: expression
```

Example:

```
lambda x: x * 5
```

This is a function that takes 1 argument **`x`** and returns **`x * 5`**

## Lambda Functions - example

Can store lambdas into a variable.

Can use that variable as if it was the lambda

```
multiple_of_five = lambda x: x * 5
print(multiple_of_five(10)) #50
```

Note: while we do not usually store lambda functions in a variable, it isn’t uncommon.

**Lambda functions aren’t multi-line!**

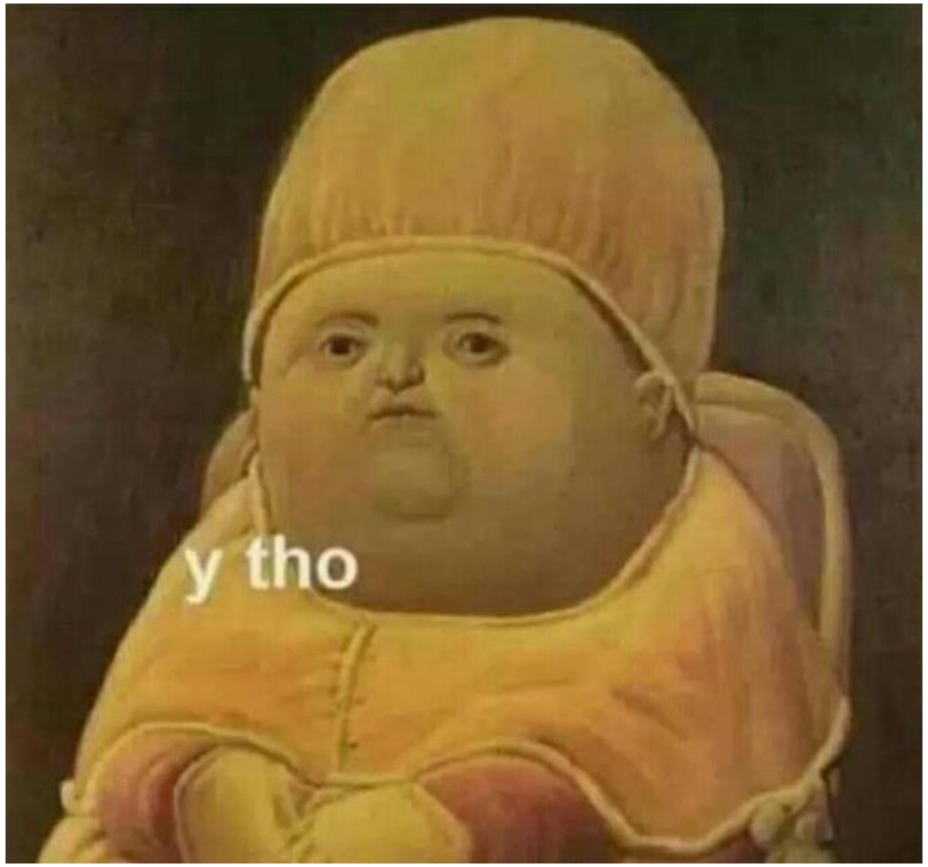
## When do we use lambda functions?

- To define callbacks.
- Asynchronous code
  i.e. code that does not run sequentially.

For example, sending an HTTP request and then executing some code when the response is received.

The send and response are considered asynchronous functions. They don’t run in sequence and there may be a wait time.

- In list comprehensions/generators
- With *filter* and *map* built-in functions



# Lambda Functions in use.

In a system far, far away….

We have a function:

```
def send_data_to_backend(protocol, data, response_callback):
     # do fancy web code here
     # execute response_callback when done
     response_callback(response)
```

Somewhere else in our system:

```
def send_analytics(user_data):
     send_data_to_backend("http", user_data, lambda response: print(response))
```

We can use lambda functions and avoid writing a new function definition.

We create an anonymous function and pass it as an argument. At some point it will get executed (once the response is received.

# Lambda Functions in use.

## Challenge Time!!

**Step 1:** Write a lambda function that takes an argument and returns the square root of the argument.

**Step 2:** Save this in a variable.

**Step 3:** Now use this lambda function in a list comprehension to create a new list which contains the square roots of each number in the list [1, 2, 3, 4, 5].

## map(function, sequence)

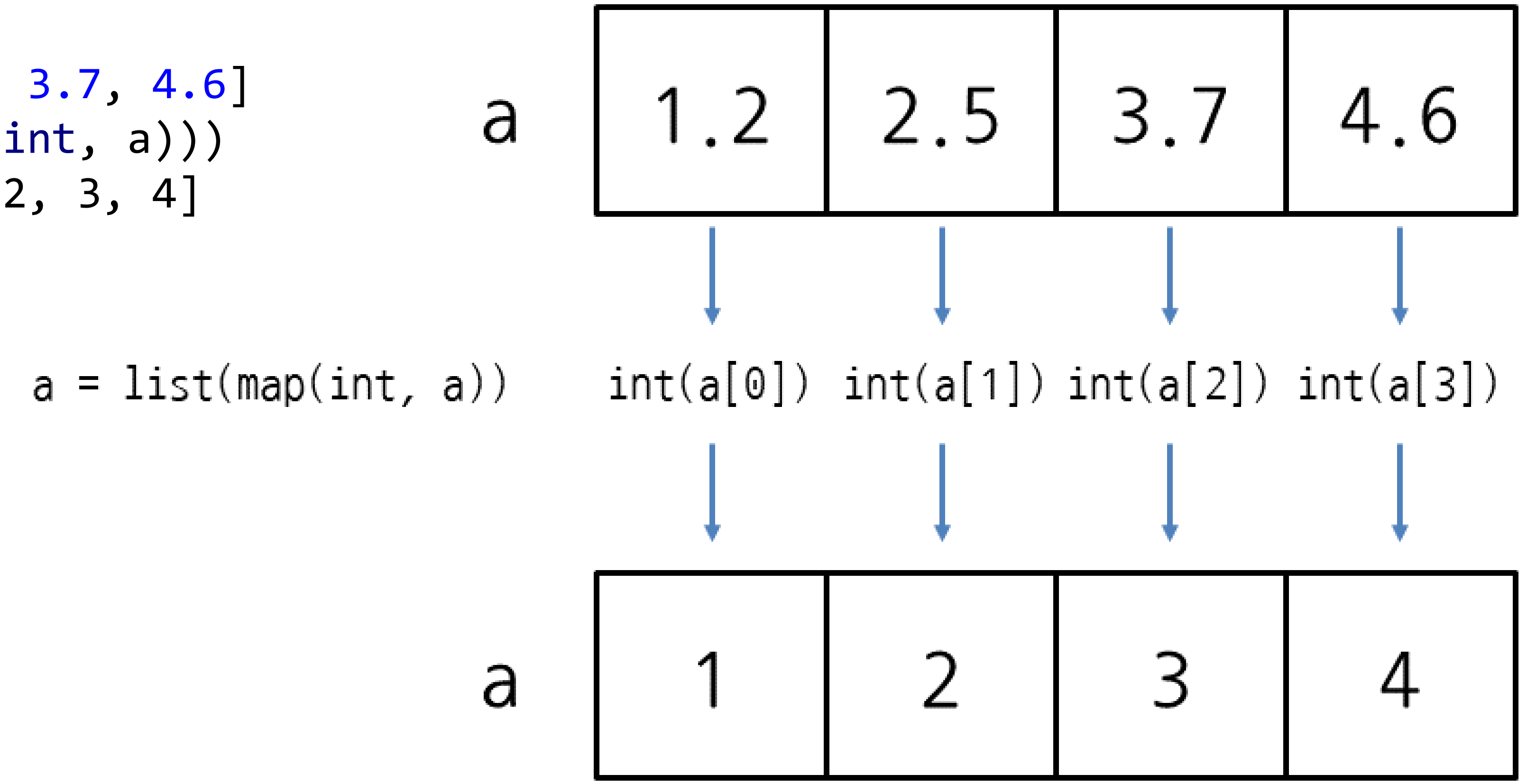
- An alternative to list comprehensions.
- The map function takes 2 arguments. A function (let’s say f(x) ) and a sequence type (let’s call this seq).
- The map function applies f(x) to each element of the seq and returns a map object.
- This map object is an iterator that represents a new sequence containing the results of applying f(x) to each element in seq

**To sum it up:**

You are *mapping* a function to each element in a sequence and returning the resultant sequence via an iterator.

## map(int, [1.2, 2.5, 3.7, 4.6])

```
a = [1.2, 2.5, 3.7, 4.6]
a = (list(map(int, a)))
print(a) #[1, 2, 3, 4]
```



## Map(function, sequence) – try it out.

```
full_names = [
    "Ross Geller",
    "Rachel Green",
    "Phoebe Buffay",
    "Chandler Bing",
    "Monica Geller",
    "Joey Tribbiani"]

first_name_map = map(lambda x: x.split()[0], full_names)
print(type(first_name_map))
print(list(first_name_map))
```

`Map the function x.split()[0] to every element in the full_names list`

# Map(function, sequence) – try it out.

```
full_names = [
    "Ross Geller",
    "Rachel Green",
    "Phoebe Buffay",
    "Chandler Bing",
    "Monica Geller",
    "Joey Tribbiani"]
```

| |
|---|
| Ross Geller |
| Rachel Green |
| Phoebe Buffay |
| Chandler Bing |
| Monica Gellar |
| Joey Tribbiani |

```
first_name_map = map(lambda x: x.split()[0], full_names)
print(type(first_name_map))
print(list(first_name_map))

Have the full list of first and last names
```

# Map(function, sequence) – try it out.

```
full_names = [
    "Ross Geller",
    "Rachel Green",
    "Phoebe Buffay",
    "Chandler Bing",
    "Monica Geller",
    "Joey Tribbiani"]
```

| |
|---|
| Ross Geller |
| Rachel Green |
| Phoebe Buffay |
| Chandler Bing |
| Monica Gellar |
| Joey Tribbiani |

| |
|---|
| Ross Geller |

```
first_name_map = map(lambda x: x.split()[0], full_names)
print(type(first_name_map))
print(list(first_name_map))

Get the first element in the list
```

## Map(function, sequence) – try it out.

```
full_names = [
    "Ross Geller",
    "Rachel Green",
    "Phoebe Buffay",
    "Chandler Bing",
    "Monica Geller",
    "Joey Tribbiani"]
```

| | |
|---|---|
| Ross | Geller |

| |
|---|
| Ross Geller |
| Rachel Green |
| Phoebe Buffay |
| Chandler Bing |
| Monica Gellar |
| Joey Tribbiani |

```
first_name_map = map(lambda x: x.split()[0], full_names)
print(type(first_name_map))
print(list(first_name_map))
```

Split the string into a list of each word

# Map(function, sequence) – try it out.

```
full_names = [
    "Ross Geller",
    "Rachel Green",
    "Phoebe Buffay",
    "Chandler Bing",
    "Monica Geller",
    "Joey Tribbiani"]
```

| |
|---|
| Ross Geller |
| Rachel Green |
| Phoebe Buffay |
| Chandler Bing |
| Monica Gellar |
| Joey Tribbiani |

| | |
|---|---|
| **Ross** | Geller |

```
first_name_map = map(lambda x: x.split()[0], full_names)
print(type(first_name_map))
print(list(first_name_map))
```

```
Get the first element of that list of words and add it to the map
```

## Map(function, sequence) – try it out.

```
full_names = [
    "Ross Geller",
    "Rachel Green",
    "Phoebe Buffay",
    "Chandler Bing",
    "Monica Geller",
    "Joey Tribbiani"]

#these two lines both extract the first names from full_names
first_name_comprehension = [x.split()[0] for x in full_names]
first_name_map = map(lambda x: x.split()[0], full_names)

#Notice how with map you don’t need to use a for loop. The expression is
automatically applied to all members in the list
```

## Map(function, seq_args_1, seq_args2, …, seq_args_n)

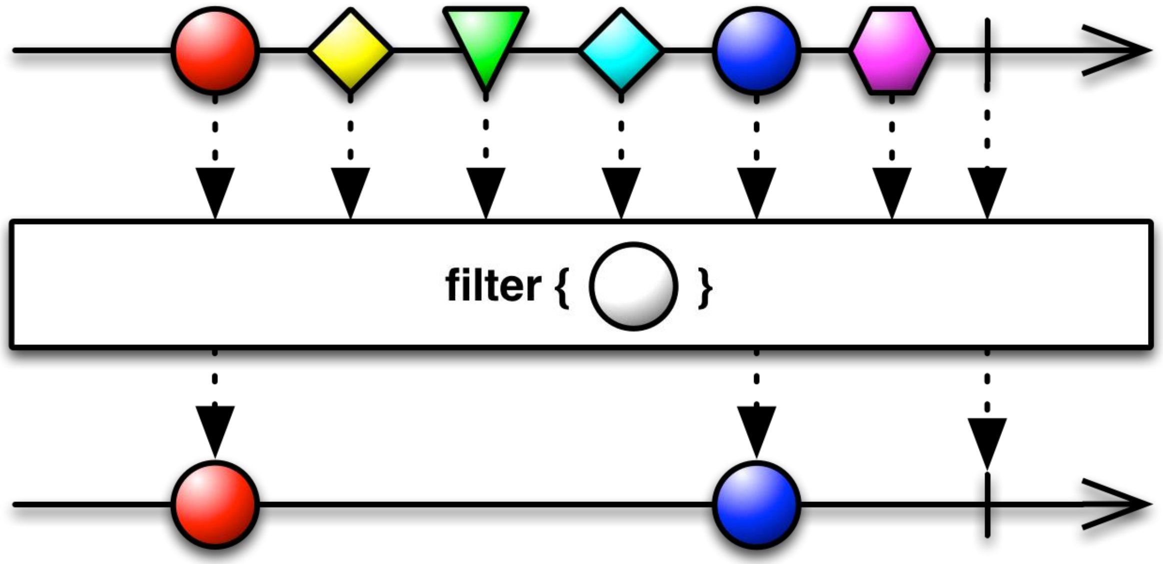
What if we have a lambda function that takes multiple arguments?

```
lambda x, y: f"{x} {y}"
```

```
first_names = ["Ross", "Rachel", "Pheobe", "Chandler", "Monica","Joey"]
last_names = ["Geller", "Green", "Buffay", "Bing", "Geller","Tribbiani"]

full_names_map = map(lambda x, y: f"{x} {y}", first_names, last_names)

print(type(full_names_map))
print(tuple(full_names_map))
```

## filter(function, sequence)

- Does exactly what the name suggests. Filters a number of elements from a given sequence.
- The first argument ‘function’ needs to be a function that returns True or False.
- This function is then applied to each element of the second argument, a sequence.
- If the result is false, that element is skipped, otherwise it is included in the resultant filter object. This is an iterator similar to the map object.
- *filter()* can only work with one argument and one sequence.



## Filter(function, sequence) – Example

Say we want to do the most cliché example.

Let’s create a filter that extracts all the even numbers from a list with numbers 1 to 10.

Notice how the lambda does not return the true/false value

```
num_list = list(range(1, 10))
even_nums_filter = filter(lambda x: x % 2 == 0, num_list)
print(type(even_nums_filter)) #<class 'filter'>
print(list(even_nums_filter)) #[2, 4, 6, 8]
```

## Filter(function, sequence) – Example

Say we want to do the most cliché example.

Let’s create a filter that extracts all the even numbers from a list with numbers 1 to 10.

The same as before but using a regular function instead of lambda. Notice how we must return the true/false value when using a function

```
def even(x):
    return x % 2 == 0

num_list = list(range(1, 10))
even_nums_filter = filter(even, num_list)
print(type(even_nums_filter)) #<class 'filter'>
print(list(even_nums_filter)) #[2, 4, 6, 8]
```

## Lambda Functions in use.

**Challenge Time!!**

**Given a string "Hello  world" and a list of vowels – ["a", "e", "i", "o", "u"], write  filter that only selects the characters that are NOT a vowel.**

```
 string = "Hello world"
even_nums_filter = ???
print(type(even_nums_filter))
print(list(even_nums_filter))
```

## A note:

Guido van Rossum is the author of the Python programming language.

He believes that Lambda functions, filter and map should be taken out of python.

Everything we can do with map and filter can be done using comprehensions and generators.

Comprehensions and generators are more in line with the zen of python than filter() and map().

Unfortunately he’s received a lot of resistance.

Lambda functions are just an integral part of a developers toolkit. You will definitely come across some filter() and map() code in the “real world”.

## The Zen of Python

Type this in your python command shell

```
>>> import this
```

# The Zen of Python

This is very important.

Now that:

- You have written some advanced code.
- You have an understanding of how object oriented design can impact the a program
- Have a number of tools in your programming tool belt

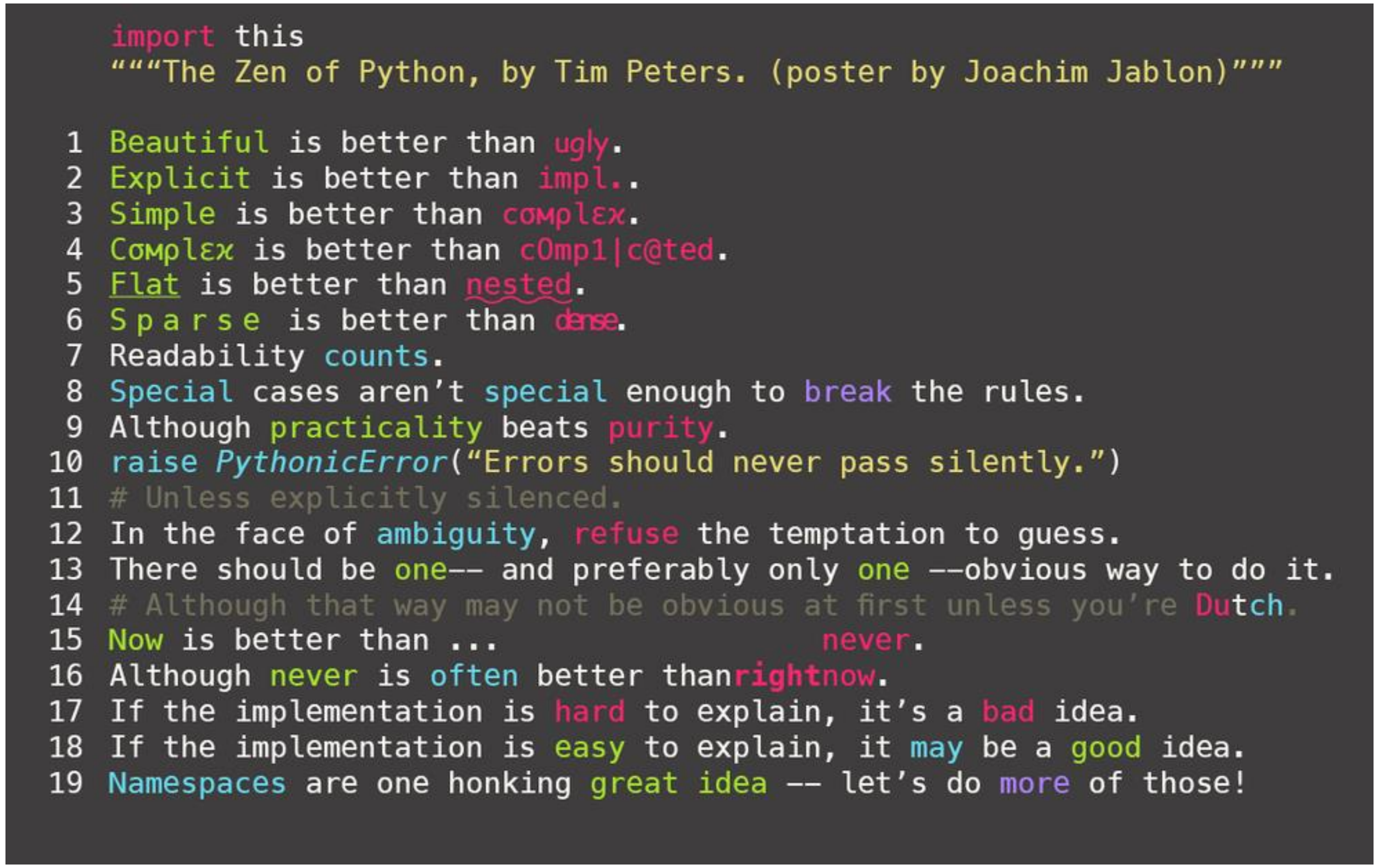
You can finally appreciate this.

This is what ‘Pythonic’ means.

Memorize this.

Make it your coding mantra.

[https://inventwithpython.com/blog/2018/08/17/the-zen-of-python-explained/](https://inventwithpython.com/blog/2018/08/17/the-zen-of-python-explained/)



```
import this
"""The Zen of Python, by Tim Peters. (poster by Joachim Jablon)"""

Beautiful is better than ugly.
Explicit is better than impl..
Simple is better than complex.
Complex is better than c0mp1|c@ted.
Flat is better than nested.
S p a r s e  is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
raise PythonicError("Errors should never pass silently.")
# Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
# Although that way may not be obvious at first unless you're Dutch.
Now is better than ...                  never.
Although never is often better thanrightnow.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

## That’s it for today!

Quiz on Friday based on material from week 5

Midterm review next class

Submit questions to: https://forms.gle/oWdXjJb4nWhJ9fpr8

Can submit multiple questions, but be specific