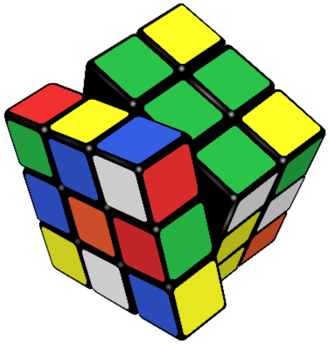
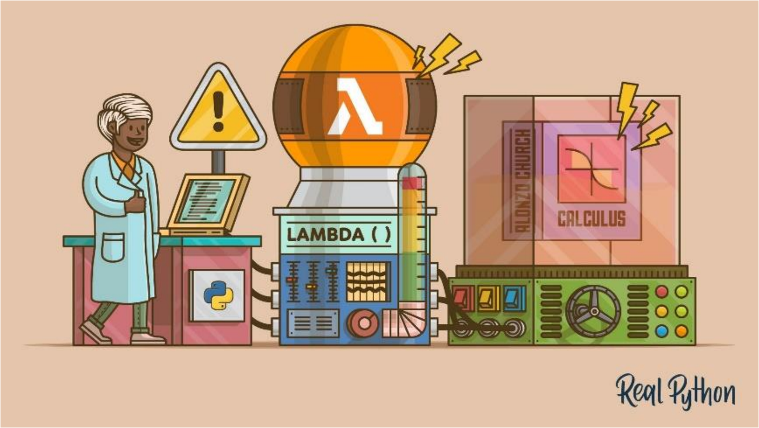




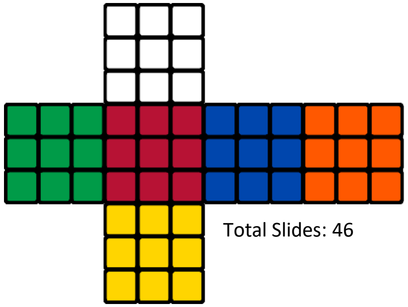
Big Data
Engineering
For Analytics

Functional Thinking in Python

Suria R Asai

(suria@nus.edu.sg)

NUS-ISS



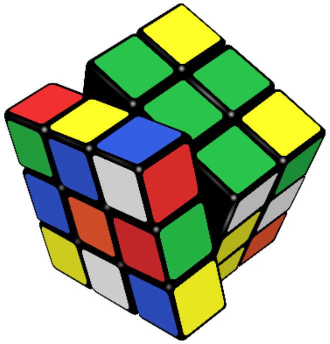
Total Slides: 46

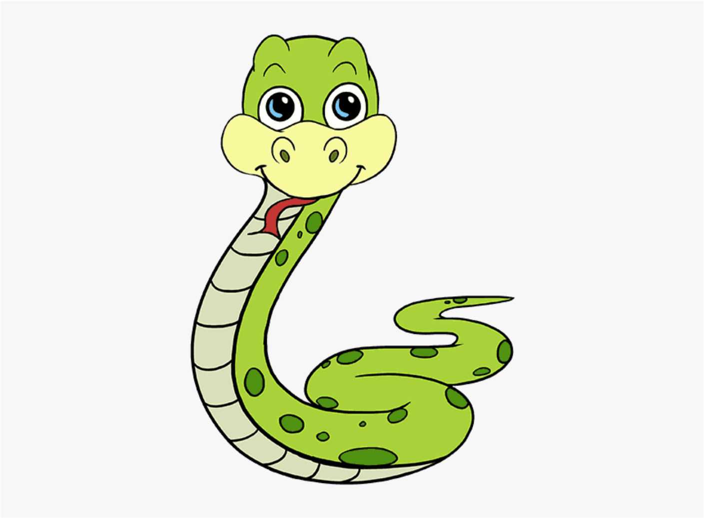
Learning Objectives

- Understand that `Python` is a multi-paradigm language; it's a mixture of procedural and functional programming.
- Understand pure functions, the concept of immutability, and referential transparency
  - **First-class and higher-order functions**, which are sometimes known as pure functions.
  - **Immutable** data.
  - Strict and non-strict **evaluation**. We can also call this eager versus lazy evaluation.
  - **Recursion** instead of an explicit loop state.
  - **Functional type systems**.

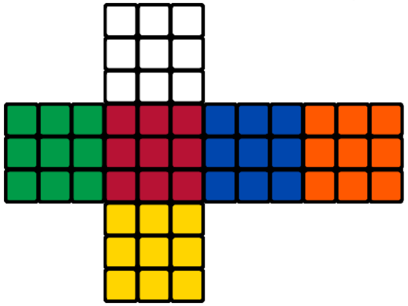
Agenda

- Basic Constructs
- Essential Functional Concepts
  - Immutable Data Set
  - Filter, Map, Currying and Reduce
  - Lambda
  - Recursion
- Higher Order Functions
- Data and Compute
  - Tuples and Strings
  - Generators, functtools and itertools
- PyCharm IDE
- Summary





Big Data Engineering For Analytics



Basic Constructs

““Give someone a program, you frustrate them for a day; teach them how to program, you frustrate them for a lifetime.”

~ David Leinwebertis

Literals and Keywords

- Every **object** has an identity, a type and a value.
- Literals:
  - Numeric literals: **integers**, **floating** point, and **imaginary** numbers (`numbers.Number`)
  - String, Byte and Formatted String Literals
- Keywords & Constants

```
False      class      finally    is         return
None       continue   for        lambda     try
True       def        from       nonlocal   while
and        del        global     not        with
as         elif       if         or         yield
assert     else       import     pass
break      except     in         raise
```

```
False True None NotImplemented Ellipsis quit() exit()
```

Data Types

- Python Data Types
  - Numeric
    - **Integer**
    - **Float**
    - Complex Number
  - Dictionary
  - **Boolean**
  - Set
    - **Frozen Set**
  - Sequence
    - **Strings**
    - **Tuple**
    - List

type	*set of values*	*common operators*	*sample literals*
`int`	*integers*	`+ - * // % **`	`99 12 2147483647`
`float`	*floating-point numbers*	`+ - * / **`	`3.14 2.5 6.022e23`
`bool`	*true-false values*	`and or not`	`True False`
`str`	*sequences of characters*	`+`	`'AB' 'Hello' '2.5'`

Basic built-in data types

Immutable?

Operators

- Arithmetic operators
  - addition (+), subtraction (-), multiplication (*), division (/), and remainder (%)
- Relational operators
  - ==, !=, >, <, >= and <=
- Logical operators
  - NOT, AND, OR

Operator	Result
x + y	Sum of x and y
x - y	Difference of x and y
x * y	Product of x and y
x / y	Quotient of x and y
x // y	Floored quotient of x and y
x % y	Remainder of x and y
-x	x negated
+x	x unchanged
abs(x)	Absolute value or magnitude of x
int(x)	x converted to integer
float(x)	x converted to floating point
divmod(x, y)	Returns the pair (x // y, x % y)
pow(x, y)	x to the power y
x ** y	x to the power y

Operator	Result
not x	Returns false if x is true, else false
x and y	Returns x if x is false, else returns y
x or y	Returns y if x is false, else returns x

Operator	Meaning
<	Less than or equal to
>	Greater than
>=	Greater than or equal
==	Equal to
!=	Not equal to
is	Object identity
is not	Negated object identity

Built in Functions

Big Data Engineering For Analytics

Built-in Functions				
abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	**object()**	**sorted()**
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	**filter()**	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	**frozenset()**	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	**map()**	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

Scope

- global
- nonlocal

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

Output

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

Simple Statements

- Expression statements
- Assignment statements
- **assert** statement
- **pass** statement
- **del** statement
- **return** statement
- **yield** statement
- **raise** statement
- **break** statement
- **continue** statement
- **import** statement
- **global** statement
- **nonlocal** statement

Compound Statements – If

Big Data Engineering For Analytics

- The if, while and for statements implement traditional control flow constructs.

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

Compound Statements – Loops

```
>>> # Measure some strings:
... words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

```
range(5, 10)
   5, 6, 7, 8, 9

range(0, 10, 3)
   0, 3, 6, 9

range(-10, -100, -30)
  -10, -40, -70
```

```
>>> # Fibonacci series:
... # the sum of two elements defines the next
... a, b = 0, 1
>>> while b < 10:
...     print(b)
...     a, b = b, a+b
...
1
1
2
3
5
8
```

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...     else:
...         # loop fell through without finding a factor
...         print(n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

Data Type Libraries

- datetime — Basic date and time types
- calendar — General calendar-related functions
- collections — Container datatypes
- collections.abc — Abstract Base Classes for Containers
- heapq — Heap queue algorithm
- bisect — Array bisection algorithm
- array — Efficient arrays of numeric values
- weakref — Weak references
- types — Dynamic type creation and names for built-in types
- copy — Shallow and deep copy operations
- pprint — Data pretty printer
- reprlib — Alternate repr() implementation
- enum — Support for enumerations

Numeric Libraries and Others

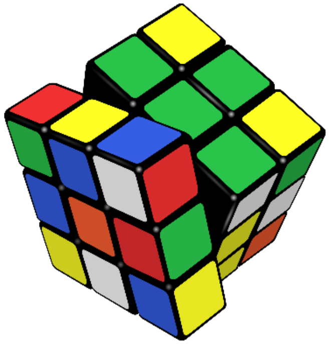
- Numeric and Mathematical Modules
  - numbers — Numeric abstract base classes
  - math — Mathematical functions
  - cmath — Mathematical functions for complex numbers
  - decimal — Decimal fixed point and floating point arithmetic
  - fractions — Rational numbers
  - random — Generate pseudo-random numbers
  - statistics — Mathematical statistics functions
- File and Directory Access, OS, Development & Interpreter Services
- Data Persistence, Data Compression & Archive Services
- File Formats, Cryptographic Services
- Concurrency, IPC, Internet, Markup Support Services
- Multimedia, UI, I18n, Performance and Runtime Services

Class and Instance Variables

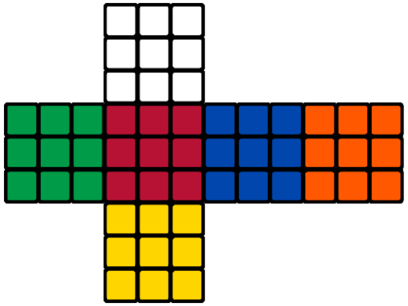
- instance variables are for data unique to each instance
- class variables are shared by all instances

```
class Dog:

    kind = 'canine'     # class var shared by all instances

    def __init__(self, name):
        self.name = name
        self.tricks = []  # instance var creates a new empty list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)
```

```
>>> d = Dog('Fido')
>>> e = Dog('Dido')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']
>>>d.kind
'canine'
```



Big Data Engineering For Analytics



Essential Functional Concepts

"While functions being unable to change state is good because it helps us reason about our programs, there's one problem with that. If a function can't change anything in the world, how is it supposed to tell us what it calculated?"

~Miran Lipovaca

Turtles - all the way down!!!

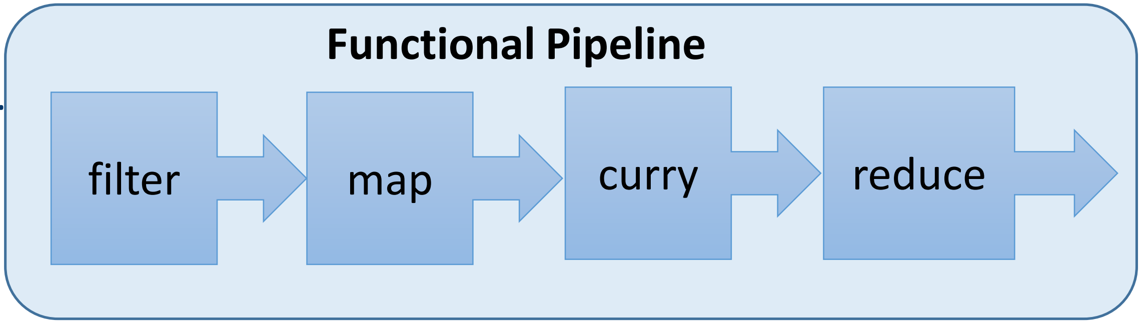
- Our functional Python programs will rely on the following three stacks of abstractions:
  - ➢ Our applications will be functions—all the way down—until we hit the objects
  - ➢ The underlying Python runtime environment that supports our functional programming is objects—all the way down—until we hit the libraries
  - ➢ The libraries that support Python are a turtle on which Python stands

All programming languages rest on abstractions, libraries, frameworks and virtual machines. These abstractions, in turn, may rely on other abstractions, libraries, frameworks and virtual machines.
The most apt metaphor is this: the world is carried on the back of a giant turtle. The turtle stands on the back of another giant turtle. And that turtle, in turn, is standing on the back of yet another turtle.
It's turtles all the way down.
- Anonymous

Turtles All the Way Down

Imperative vs Functional Constructs

- First-class and higher-order functions, aka **pure functions** are succinct and expressive.
  - ➢Practical **Benefits:**
    - Distributed(Big) Datasets.
    - Parallel Processing.
    - Formal provability.
    - Modularity.
    - Composability.
    - Ease of debugging / testing.
  - ➢Prefers **Immutable data**.
  - ➢Strict and non-strict **evaluation**. We can also call this eager versus lazy evaluation.
  - ➢**Recursion** instead of an explicit loop state.
  - ➢Uses **Functional type systems**.
  - ➢**A Python lambda is a pure function.**
    - While this isn't a highly recommended style, it's possible to create pure functions through the lambda objects.



Simple Pure Functions

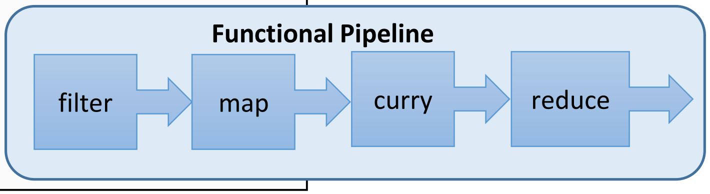
- No Side effects, return value only, shallow copy

Functional programming is a way of writing software applications using only pure functions and immutable values.


```python
# Python program to demonstrate pure functions
# A pure function that does not change the input list and
# returns the new List
def pure_func(List):

        New_List = []

        for i in List:
                New_List.append(i**2)

        return New_List

# Driver's code
Original_List = [1, 2, 3, 4]
Modified_List = pure_func(Original_List)

print("Original List:", Original_List)
print("Modified List:", Modified_List)
```

Output Input

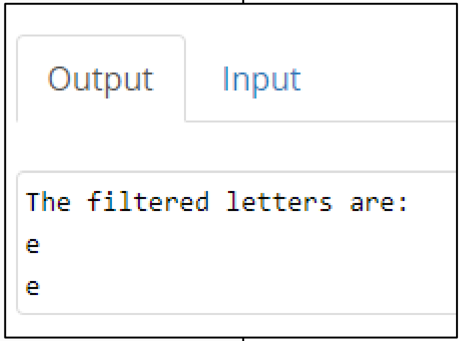
```
Original List: [1, 2, 3, 4]
Modified List: [1, 4, 9, 16]
```

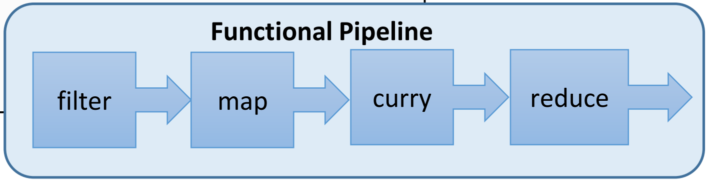


Filtering Example

Filter is a generator expression that returns an iterator from a sequence that meet a certain condition (predicate).

```
# Python program to demonstrate working of filter.
# function that filters vowels
def fun(variable):

        letters = ['a', 'e', 'i', 'o', 'u']

        if (variable in letters):
                return True
        else:
                return False
# sequence
sequence = ['g', 'e', 'e', 'j', 'k', 's', 'p', 'r']
# using filter function
filtered = filter(fun, sequence)
print('The filtered letters are:')
for s in filtered:
        print(s)
```

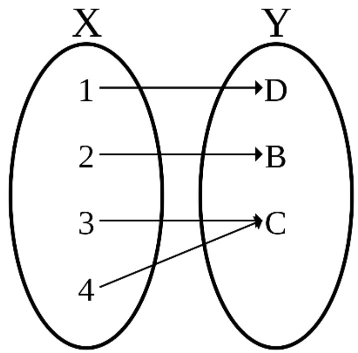


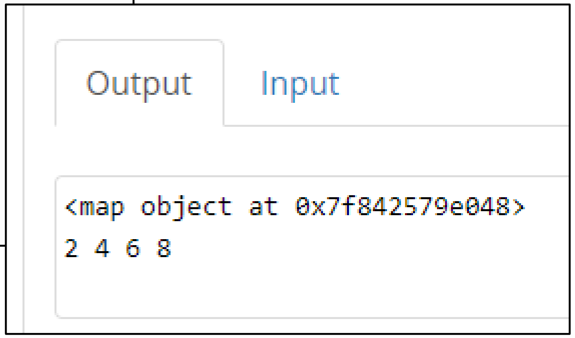


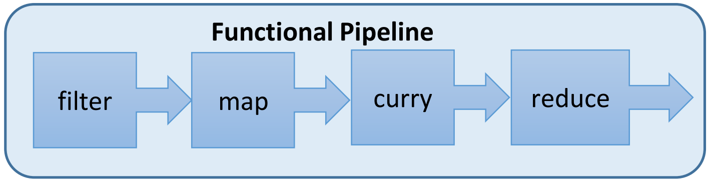
Map Example

Map is a generator expression that returns an iterator over a sequence.

```
# Python program to demonstrate working of map.
# Return double of n
def addition(n):
        return n + n

# We double all numbers using map()
numbers = (1, 2, 3, 4)
results = map(addition, numbers)
# Does not print the value
print(results)
# For printing value
for result in results:
        print(result, end = " ")
```



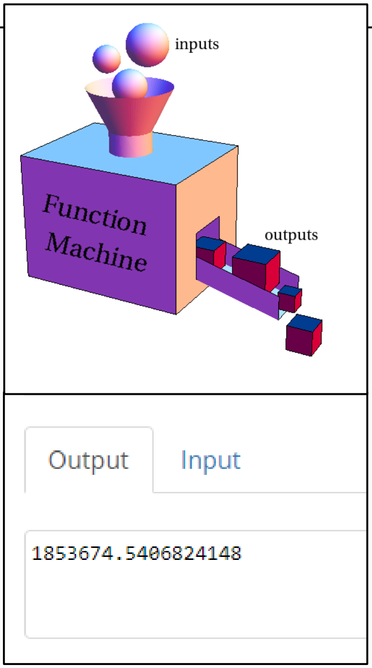


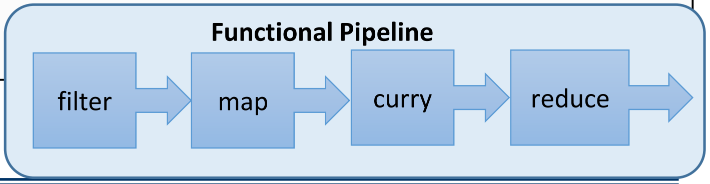


Currying Example

Currying is a function transforms multiple function into execution of sequential functions.

```
# Demonstrate Currying of composition of function
def change(b, c, d):
        def a(x):
                return b(c(d(x)))
        return a
def kilometer2meter(dist):
        """ Function that converts km to m. """
        return dist * 1000
def meter2centimeter(dist):
        """ Function that converts m to cm. """
        return dist * 100
def centimeter2feet(dist):
        """ Function that converts cm to ft. """
        return dist / 30.48
if __name__ == '__main__':
        transform = change(centimeter2feet, meter2centimeter, kilometer2meter )
        e = transform(565)
        print(e)
```





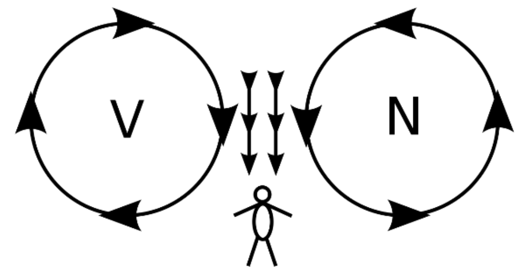
Reduce Example

Reduce is a function that accepts a function and a sequence and returns a single value calculated cumulatively.

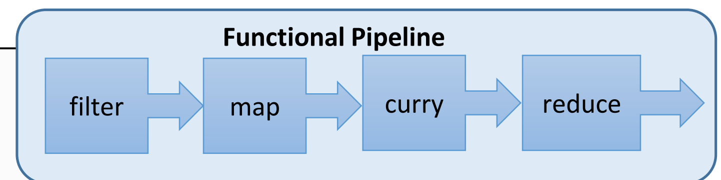
```
from functools import reduce

def do_sum(x1, x2):
    return x1 + x2

print(reduce(do_sum, [1, 2, 3, 4]))
```

Output Input

```
10
```




```
import functools
import operator
# initializing list
lis = [ 1 , 3, 5, 6, 2, ]
# using reduce and operator
print ("The sum of the list elements is : ",end="")
print (functools.reduce(operator.add,lis))
print ("The product of list elements is : ",end="")
print (functools.reduce(operator.mul,lis))
print ("The concatenated product is : ",end="")
print (functools.reduce(operator.add,["geeks","for","geeks"]))
```



Output Input

```
The sum of the list elements is : 17
The product of list elements is : 180
The concatenated product is : geeksforgeeks
```

OOP to Recursion

- In a functional sense, the sum of the multiples of three and five can be defined in two parts:
  - The sum of a sequence of numbers
  - A sequence of values that pass a simple test condition, for example, being multiples of three and five
- The sum of a sequence has a simple, recursive definition:

```
class Summable_List(list):
    def sum(self):
        s = 0
        for v in self:
            s += v
        return s
```

```
def sumr(seq):
    if len(seq) == 0: return 0
    return seq[0] + sumr(seq[1:])
```

- Similarly, a sequence of values can have a simple, recursive definition, as follows:

```
def until(n, filter_func, v):
    if v == n: return []
    if filter_func(v): return [v] + until(n, filter_func, v+1)
    else: return until(n, filter_func, v+1)
```

Lambda is simple

- Lambdas are used to emphasize succinct definitions of simple functions.
  - ➢ Anything more complex than a one-line expression requires the def statement.

$$\{n | 1 \leq n < 10 \wedge (n \mod 3 = 0 \vee n \mod 5 = 0\}$$

```
mult_3_5 = lambda x: x%3==0 or x%5==0
print(mult_3_5(3))
print(mult_3_5(4))
print(mult_3_5(5))
```

Output

```
True
False
True
```

  - ➢ Lambdas can only contain a single Python expression.
  - ➢ A lambda expression can have any number of arguments (including none), for example:

```
lambda: 1   # No arguments
lambda x, y: x + y
lambda a, b, c, d: a*b + c*d
```

  - ➢ Function as return value

```
def add1():
    return lambda x: x + 1

f = add1()
print(f(2))
```

Output

```
3
```

Big Data
Engineering
For Analytics

Factorial Example

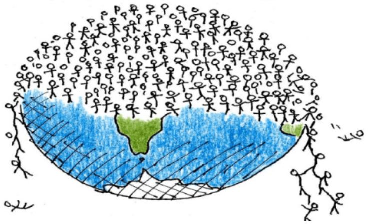
- A slightly less trivial example, factorial in recursive and iterative style:

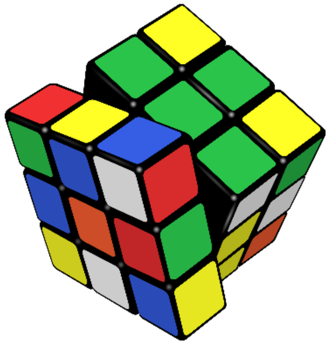
```
def factorialI(N):
    "Iterative factorial function"
    assert isinstance(N, int) and N >= 1
    product = 1
    while N >= 1:
        product *= N
        N -= 1
    return product
```

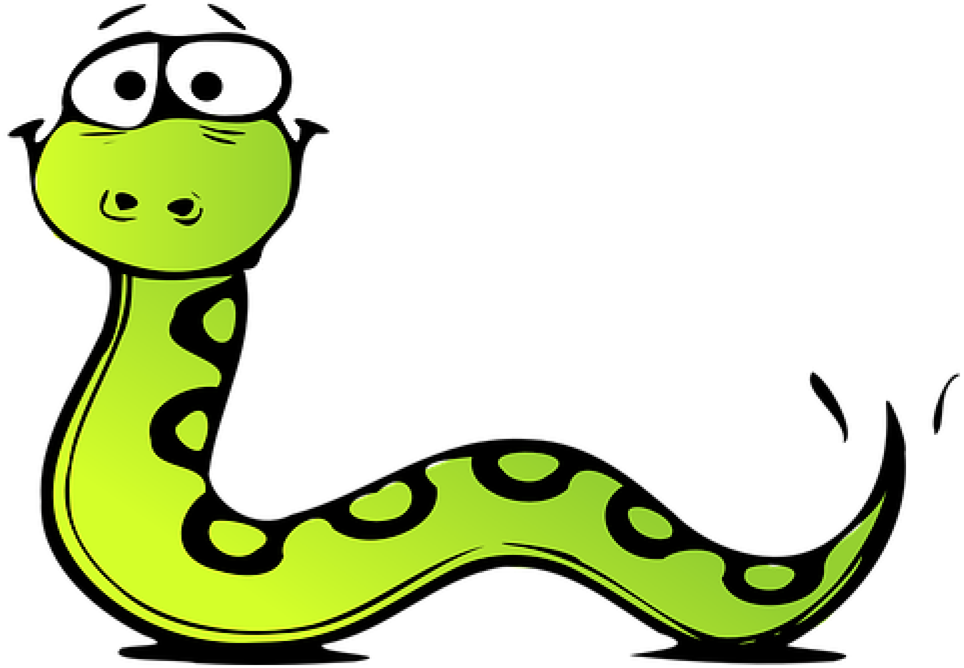
```
def factorialR(N):
    "Recursive factorial function"
    assert isinstance(N, int) and N >= 1
    return 1 if N <= 1 else N * factorialR(N-1)
```

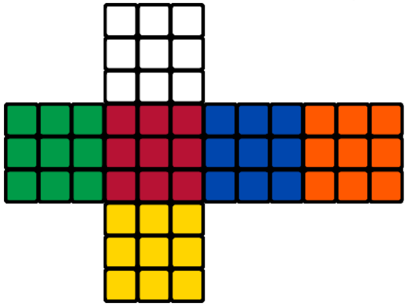
- A higher order function implementation from libraries

```
from functools import reduce
from operator import mul
def factorialHOF(n):
    return reduce(mul, range(1, n+1), 1)
```









Data & Compute

"In order to understand recursion, one must first understand recursion. (Anonymous)"

Functions are first class objects

- Python encourages to focus on **tuples,** named tuples and **immutable** collections such as strings
- Python **imposes a recursion limit**, and doesn't automatically handle Tail Call Optimization (TCO)
  - We must optimize recursions manually using a generator expression.
  - Use generator and generator expressions to work with collection of objects
- **Generator expressions** will perform the following tasks:
  - Conversions
  - Restructuring
  - Complex calculations

strings

Big Data Engineering For Analytics

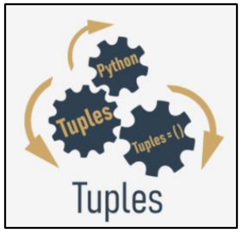
- Python strings are immutable
  - str object has a number of methods to manipulate strings and they are pure functions
  - The syntax for str method functions is postfix, where most functions are prefix.
  - For example, in this expression, **`len(variable.title())`**, the **`title()`** method is in postfix notation and the **`len()`** function is in prefix notation.
- A simple prefix function to strip punctuation

```
def remove(str: Text, chars: Text) -> Text:
    if chars:
        return remove(
            str.replace(chars[0], ""),
            chars[1:]
        )
    return str
```

Tuples and Named Tuples

- Python **tuples** are immutable objects, suitable for functional programming.
  - ➢A tuple has very few method functions, using prefix syntax.
  - ➢There are a number of use cases for tuples, particularly when working with *list-of-tuple*, *tuple-of-tuple*, and *generator-of-tuple* constructs.
  - ➢The **namedtuple** class adds an essential feature to a tuple: a name that we can use instead of an index.

```
red = lambda color: color[0]green = lambda color:
color[1]blue = lambda color: color[2]

from typing import Tuple, Callable
RGB = Tuple[int, int, int]
red: Callable[[RGB], int] = lambda color: color[0]
```




```
from typing import NamedTuple
class Color(NamedTuple):
     """An RGB color."""
     red: int
     green: int
     blue: int
     name: str
```

Generator Expressions

- A generator expression (iterable) is **lazy** and creates objects only as required; this can improve performance.
  - Two important caveats on generator expressions, as follows:
    - Generators appear to be sequence-like. The few exceptions include using a function such as the len() function that needs to know the size of the collection.
    - Generators can be used only once. After that, they appear empty.

```
def pfactorsl(x: int) -> Iterator[int]:
    if x % 2 == 0:
        yield 2
        if x//2 > 1:
            yield from pfactorsl(x//2)
        return
    for i in range(3, int(math.sqrt(x)+.5)+1, 2):
        if x % i == 0:
            yield i
            if x//i > 1:
                yield from pfactorsl(x//i)
            return
    yield x
```

Recursive generators

- In a recursive generator function, the return statement is tricky.
  - Do not use the following command line:

```
return recursive_iter(args)
```

```
for result in recursive_iter(args):
    yield result
yield from recursive_iter(args)
```

  - It returns only a generator object; it doesn't evaluate the function to return the generated values. Use any of the following:

```
def pfactorsr(x: int) -> Iterator[int]:
    def factor_n(x: int, n: int) -> Iterator[int]:
        if n*n > x:
            yield x
            return
        if x % n == 0:
            yield n
            if x//n > 1:
                yield from factor_n(x//n, n)
        else:
            yield from factor_n(x, n+2)
    if x % 2 == 0:
        yield 2
        if x//2 > 1:
            yield from pfactorsr(x//2)
        return
    yield from factor_n(x, 3)
```

$$3 \leq n \leq \sqrt{x}$$

- If the candidate factor, n, is outside the range, then x is prime.
- Otherwise, we'll see whether n is a factor of x. If so, we'll yield n and all factors of x/n.
- If n is not a factor, we'll evaluate the function recursively using n+2

Type of Functions

- We need to distinguish between two broad species of functions, as follows:
  - **Scalar functions**: They apply to individual values and compute an individual result. Functions such as abs(), pow(), and the entire math module are examples of scalar functions.
  - **Collection functions**: They work with iterable collections.
- We can further subdivide the collection functions into three subspecies:
  - **Reduction**: This uses a function to fold values in the collection together, resulting in a single final value.
    - For example, if we fold (+) operations into a sequence of integers, this will compute the sum. This can be also be called an aggregate function, as it produces a single aggregate value for an input collection.
  - **Mapping**: This applies a scalar function to each individual item of a collection; the result is a collection of the same size.
  - **Filter**: This applies a scalar function to all items of a collection to reject some items and pass others. The result is a subset of the input.

itertools — Functions creating iterators for efficient looping - 1

Big Data Engineering For Analytics

- Iterator building blocks inspired by Haskell, recast to Python.
- Infinite Iterators

Iterator	Arguments	Results	Example
count()	start, [step]	start, start+step, start+2*step, …	count(10) --> 10 11 12 13 14 ...
cycle()	p	p0, p1, … plast, p0, p1, …	cycle('ABCD') --> A B C D A B C D ...
repeat()	elem [,n]	elem, elem, elem, … endlessly or up to n times	repeat(10, 3) --> 10 10 10

- Combinatorics Iterators

Iterator	Arguments	Results
product()	p, q, … [repeat=1]	cartesian product, equivalent to a nested for-loop Example product('ABCD', repeat=2) AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD
permutations()	p[, r]	r-length tuples, all possible orderings, no repeated elements Example permutations('ABCD', 2) AB AC AD BA BC BD CA CB CD DA DB DC
combinations()	p, r	r-length tuples, in sorted order, no repeated elements combinations('ABCD', 2) AB AC AD BC BD CD
combinations_with_replacement()	p, r	r-length tuples, in sorted order, with repeated elements combinations_with_replacement('ABCD', 2) AA AB AC AD BB BC BD CC CD DD

itertools — Functions creating iterators for efficient looping – 2.

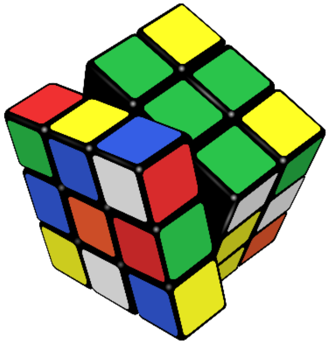
- Terminating Iterators

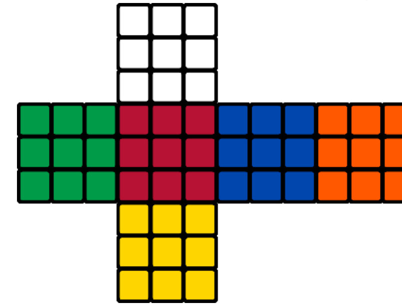
Iterator	Arguments	Results	Example
accumulate()	p [,func]	p0, p0+p1, p0+p1+p2, …	accumulate([1,2,3,4,5]) --> 1 3 6 10 15
chain()	p, q, …	p0, p1, … plast, q0, q1, …	chain('ABC', 'DEF') --> A B C D E F
chain.from_iterable()	iterable	p0, p1, … plast, q0, q1, …	chain.from_iterable(['ABC', 'DEF']) --> A B C D E F
compress()	data, selectors	(d[0] if s[0]), (d[1] if s[1]), …	compress('ABCDEF', [1,0,1,0,1,1]) --> A C E F
dropwhile()	pred, seq	seq[n], seq[n+1], starting when pred fails	dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 4 1
filterfalse()	pred, seq	elements of seq where pred(elem) is false	filterfalse(lambda x: x%2, range(10)) --> 0 2 4 6 8
groupby()	iterable[, key]	sub-iterators grouped by value of key(v)	
islice()	seq, [start,] stop [, step]	elements from seq[start:stop:step]	islice('ABCDEFG', 2, None) --> C D E F G
starmap()	func, seq	func(*seq[0]), func(*seq[1]), …	starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000
takewhile()	pred, seq	seq[0], seq[1], until pred fails	takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4
tee()	it, n	it1, it2, … itn splits one iterator into n	
zip_longest()	p, q, …	(p[0], q[0]), (p[1], q[1]), …	zip_longest('ABCD', 'xy', fillvalue='-') --> Ax By C- D-

functools

- The **`functools`** module is for higher-order functions: functions that act on or return other functions.
  - functools.**cmp_to_key**(func): Transform an old-style comparison function to a key function.
  - @functools.**lru_cache**(maxsize=128, typed=False) Decorator to wrap a function with a memoizing callable that saves up to the maxsize most recent calls.
  - @functools.**total_ordering** Given a class defining one or more rich comparison ordering methods, this class decorator supplies the rest.
  - functools.**partial**(func, *args, **keywords) Return a new partial object which when called will behave like func called with the positional arguments args and keyword arguments keywords.
  - functools.**reduce**(function, iterable[, initializer]) Apply function of two arguments cumulatively to the items of sequence, from left to right, so as to reduce the sequence to a single value.

Big Data Engineering For Analytics





Higher Order Functions

"Object oriented programming makes code understandable by encapsulating moving parts. Functional programming makes code understandable by minimizing moving parts."

~ Michael Feathers

Higher Order Functions

A higher-order function is simply a function that takes one or more functions as arguments and/or produces a function as a result.

- higher-order functions provide building blocks to express complex concepts by combining simpler functions into new functions.
  - They allow chaining and combining higher-order functions

```
def compose(*funcs):
    """Return a new function s.t.
       compose(f,g,...)(x) == f(g(...(x)))"""
    def inner(data, funcs=funcs):
        result = data
        for f in reversed(funcs):
            result = f(result)
        return result
    return inner

times2 = lambda x: x*2
minus3 = lambda x: x-3
mod6 = lambda x: x%6
f = compose(mod6, times2, minus3)
print(all(f(i)==((i-3)*2)%6 for i in range(1000000)))
```

Output

```
True
```

Types of Higher order functions

Big Data Engineering For Analytics

- There are three varieties of higher-order functions as follows:
  - Functions that accept functions as one (or more) of their arguments
  - Functions that return a function
  - Functions that accept a function and return a function, a combination of the preceding two features
- The max() and min() functions each have a dual life.
  - They are simple functions that apply to collections. They are also higher-order functions.

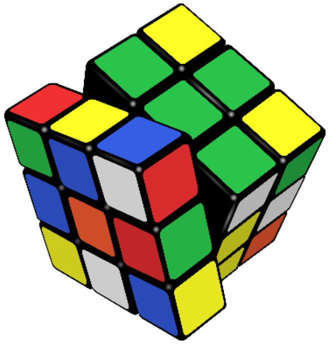
```
print(max(1, 2, 3))
print(max((1,2,3,4)))
```

Output

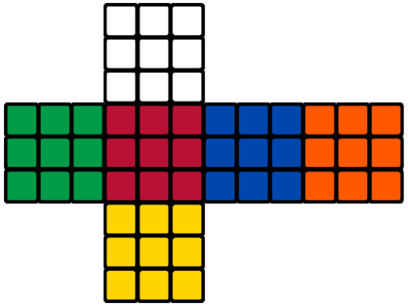
```
3
4
```

```
t = """2    3    5    7    11    13    17    19    23
29   31   37   41   43   47   53   59   61   67   71
73   79   83   89   97   101  103  107  109  113
127  131  137  139  149  151  157  163  167  173
179  181  191  193  197  199  211  223  227  229"""
data = list(v for line in t.splitlines()
         for v in line.split())
print(list(map(int, data)))
```

```
from tripdata import (
    float_from_pair, lat_lon_kml, limits, haversine, legs
)
path =
float_from_pair(float_lat_lon(row_iter_kml(source)))
trip = tuple(
    (start, end, round(haversine(start, end), 4))
        for start, end in legs(iter(path)))

long = max(dist for start, end, dist in trip)
short = min(dist for start, end, dist in trip)

print(long)
print(short)
```



Big Data Engineering For Analytics

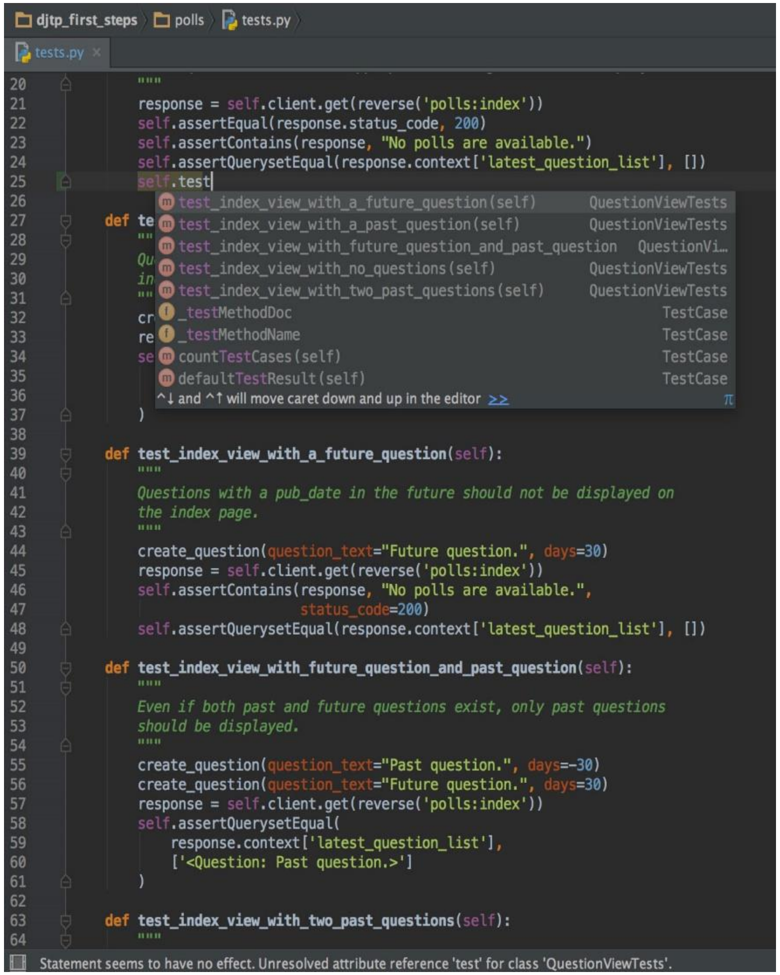


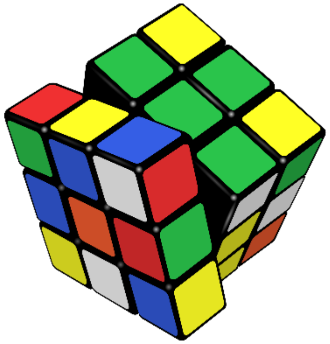
PyCharm

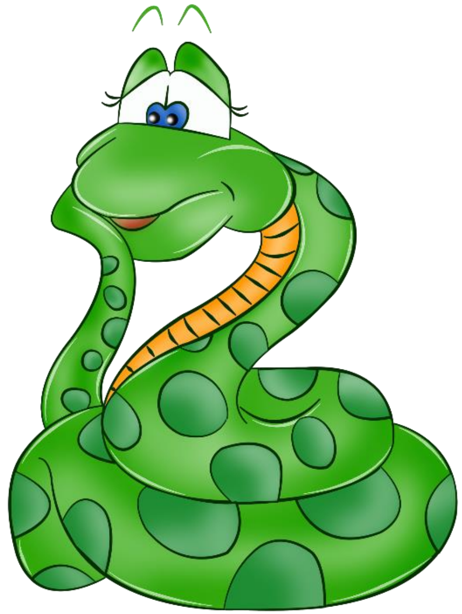
"Don't worry if it doesn't work right. If everything did, you'd be out of a job. (Mosher's Law of Software Engineering)"

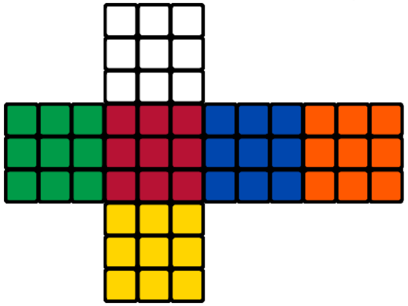
Python Tools

- PyCharm provides smart code completion, code inspections, on-the-fly error highlighting and quick-fixes, along with automated code refactorings and rich navigation capabilities.
- PyCharm offers great framework-specific support for modern web development frameworks such as Django, Flask, Google App Engine, Pyramid, and web2py.
- PyCharm supports JavaScript, CoffeeScript, TypeScript, Cython, SQL, HTML/CSS, template languages, AngularJS, Node.js, and more.
- Run, debug, test, and deploy applications on remote hosts or virtual machines, with remote interpreters, an integrated ssh terminal, and Docker and Vagrant integration.









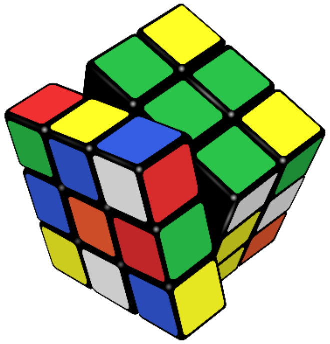
Summary

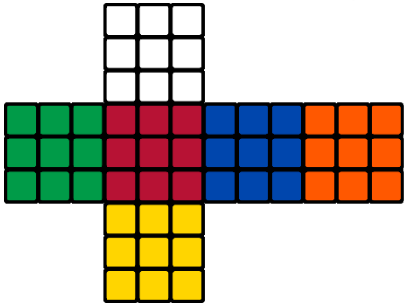
"If builders built buildings the way programmers wrote programs, then the first woodpecker that came along would destroy civilization."

~Gerald Weinberg

In Essence

- We explored *functional programming features* of Python.
- We learnt how to craft functional constructs using **map**, **filter**, **curry** and **reduce**.
  - ➢We looked at **lambda** expressions
  - ➢We looked at **recursive** constructs
  - ➢We looked at **generator** functions and how we can use these as the backbone of functional programming.
- We examined the **built-in collection** classes to show how they're used in the functional paradigm.
- We examined Python's **collection-processing features** from a functional programming viewpoint.
- We looked at **higher-order functions**: functions that accept functions as arguments as well as returning functions.





References

"The best performance improvement is the transition from the nonworking state to the working state."

~J. Osterhout

Big Data Engineering For Analytics

Official References

- Python Language Specification
- Python Documentation
- Python Functional Programming
- Python Libraries
- Python Tutorials
- Header

Books You May Enjoy. . .

- Functional Python Programming - Second Edition, by Steven F. Lott, Published by Packt Publishing, 2018
- Functional Programming in Python by Martin McBride Published by Packt Publishing, 2019
- Functional Programming in Scala By Paul Chiusano and Rúnar Bjarnason, 2014
- Mastering Functional Programming by Anatolii Kmetiuk, Packt Publishing, August 2018, ISBN: 9781788620796
- Real Python Tutorials