

# Functional Python

Python enters the 21<sup>st</sup> century





# Iterators

- An *iterator* is something that returns values one at a time
- An *iterable* is something that has values and can be iterated over
- For example, a list is an iterable, and the `iter` function will give you an iterator for it:

- ```
>>> it = iter([1, 2, 5])  
>>> next(it)  
1
```

- ```
>>> next(it)  
2
```

- ```
>>> next(it)  
5
```

- ```
>>> next(it)  
Traceback (most recent call last):  
  File "<pyshell#39>", line 1, in <module> next(it)  
StopIteration
```



# More iterables

- Other things that can be iterated over are sets, tuples, strings, dictionaries, ranges, and files:
  - ```
>>> next(iter({'one', 'two', 'three', 'four'})) # set  
'four'
```
  - ```
>>> next(iter(('one', 'two', 'three', 'four'))) # tuple  
'one'
```
  - ```
>>> next(iter('one two three four')) # string  
'o'
```
  - ```
>>> next(iter({'one': 'two', 'three': 'four'})) # dictionary  
'three'
```
  - ```
>>> next(iter(range(5, 10))) # range  
5
```
  - ```
>>> next(iter(open('testfile.txt', 'r')))  
'This is the first line of the file\n'
```
- The familiar `for` loop uses iterators
- Whenever you say `for i in X:`, `X` must be an iterable



# range

- In Python 2, `range` returned a list:
  - ```
>>> range(1, 5)
[1, 2, 3, 4]
>>> type(range(1, 5))
<type 'list'>
```
- In Python 3, `range` returns a “range” iterator:
  - ```
>>> range(0, 4)
range(0, 4)
>>> type(range(0, 4))
<class 'range'>
```
- `range` continues to work just as it used to in `for` statements
- `range` no longer works where a *list* is required



# yield

- The `yield` statement enables the use of *coroutines*
- `yield` is used like `return`, except that:
  1. You can come back to the function and continue from where you “yielded”
  2. The state of local variables is retained
- If a function *fun* contains `yield` statements, then:
  - Calling *fun* returns an *iterator*, not an ordinary value
  - To get each value of the function, call `next(fun)` as many times as desired
  - The iterator can be used in a `for` loop (just like `range` is used)

# yield example: divisors I

- ```
>>> def divisors(n):  
    for d in range(2, n):  
        if n % d == 0:  
            yield d
```
- ```
>>> for i in divisors(12):  
    print(i, end="  ")
```

2 3 4 6



# divisors II

- `>>> it = divisors(10)`
- `>>> it`  
`<generator object divisors at 0x106e1a360>`
- `>>> next(it)`  
`2`
- `>>> next(it)`  
`5`
- `>>> next(it)`  
`Traceback (most recent call last):`  
 `File "<pyshell#97>", line 1, in <module>`  
 `next(it)`  
`StopIteration`



# The Collatz sequence I

- Collatz is a recursive function, defined as follows:

```
def collatz(n):  
    if n == 1:  
        return 1  
    elif n % 2 == 0:  
        return collatz(n // 2)  
    else:  
        return collatz(3 * n + 1)
```

- When this function terminates, it will return **1**

```
>>> collatz(7)  
1
```

- We don't get to see the intermediate values, unless we modify the function significantly to save the values in a list, or something like that



# The Collatz sequence II

- We can define a Collatz *iterator*, like so:

```
>>> def collatz(n):  
    yield n  
    while n != 1:  
        if n % 2 == 0:  
            n = n // 2  
        else:  
            n = 3 * n + 1  
    yield n
```

```
>>> for i in collatz(7):  
    print(i, end=" ")
```

7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1



# Literal functions

- Here is a conventional function definition:

```
def average(x, y):  
    return (x + y) / 2
```

- Here is the same thing written as a lambda expression and assigned to a variable

```
average = lambda x, y: (x + y) / 2
```

- There is no "return"
- The part after the colon must be a single expression to be evaluated
- The value of the expression is the value returned by the function
- A lambda expression is a function, so you can call it directly,

```
>>> (lambda x, y: (x + y) / 2)(5, 10)  
7.5
```

but that's kind of silly



# Typical use of lambdas

- A lambda by itself is an “anonymous function”, that is, a function without a name
- Many functions take functions as arguments, and these arguments are often given as lambdas
- Consider: In Unicode, capital letters precede lowercase letters.  
Thus:
  - ```
>>> lst = ['Betsy', 'goes', 'to', 'the', 'University',  
          'of', 'Pennsylvania']
```
  - ```
>>> sorted(lst)  
['Betsy', 'Pennsylvania', 'University', 'goes', 'of',  
 'the', 'to']
```
- But we can sort on the lowercased (or uppercased) versions of the words
  - ```
>>> sorted(lst, key = lambda x: x.lower())  
['Betsy', 'goes', 'of', 'Pennsylvania', 'the', 'to',  
 'University']
```



# Conditional expressions

- Since you can't do very much in a single expression, Python also provides a conditional expression
  - Syntax: *expression1 if condition else expression2*
- Despite the order in which things are written,
  - Python first evaluates the *condition*
    - if the *condition* is true, Python evaluates *expression1*
    - otherwise Python evaluates *expression2*
- Conditional expressions are not limited to lambda expressions, but may be used anywhere
- Conditional expressions have very low precedence
  - Hence, they must be enclosed in parentheses when used as part of a more complex expression
    - ```
>>> 100 + (5 if 2 > 3 else 7)
107
```



# Collatz III

- ```
>>> def collatz(n):  
    yield n  
    while n != 1:  
        n = n // 2 if n % 2 == 0 else 3 * n + 1  
        yield n
```

- ```
>>> for i in collatz(7):  
    print(i, end=" ")
```

7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1



# Higher-level functions

- A *higher-level function* is a function that either:
  - Takes a function or functions as arguments, or
  - Returns a function as its value, or
  - Both of the above
- Lambda expressions are most often used as arguments to a higher-level function
- It's easy to write your own higher-level functions:
  - ```
>>> def applyTwice(f, x):  
        return f(f(x))
```
  - ```
>>> applyTwice(lambda x: x * x, 3)  
81
```
- Python has several very useful built-in higher-level functions, but many of them return iterators (which is why you need to know about iterators!)
- Given an iterator *it*, you can get a list of the returned values by calling `list(it)`



# map

- `map(function, sequence)` applies the *function* to each element of the *sequence* (which may be a list, set, string, tuple, or dictionary), and returns an *iterator* of the result
- ```
>>> double = lambda x: 2 * x
```
- ```
>>> map(double, [1, 2, 5])  
<map object at 0x102f49c50>
```
- ```
>>> list(map(double, [1, 2, 5]))  
[2, 4, 10]
```
- ```
>>> list(map(double, {1, 2, 5}))  
[2, 4, 10]
```
- ```
# iterate on keys  
>>> list(map(double, {1: 'a', 2: 'b', 5: 'e'}))  
[2, 4, 10]
```
- ```
>>> list(map(double, (1, 2, 5)))  
[2, 4, 10]
```



# filter

- `filter(predicate, sequence)` uses the *predicate* to test each element of the *sequence*, and returns an iterator which will generate those elements that satisfy the *predicate*
- ```
>>> filter(lambda x: x % 2 == 0, range(1, 10))  
<filter object at 0x102f49ad0>
```
- ```
>>> list(filter(lambda x: x % 2 == 0, range(1, 10)))  
[2, 4, 6, 8]
```
- ```
>>> list(filter(lambda x: x % 2 == 0, {1: 'a', 2: 'b',  
3: 'c', 4: 'd'}))  
[2, 4]
```



# reduce

- `functools.reduce(binaryFunction, finiteSequence)`
  1. Applies the *binaryFunction* to the first two elements of the *finiteSequence*
  2. Repeatedly applies the *binaryFunction* to the current result and the next member of the *finiteSequence*
- ```
>>> import functools
>>> functools.reduce(lambda x, y: x - y,
[100, 1, 2, 3])
94
```
- ```
>>> (((100 - 1) - 2) - 3)
94
```
- Some iterators, such as `itertools.count()`, can return an infinite sequence of values; these cannot be reduced



# “Truthy” and “falsey”

- **Truthiness** is a quality characterizing a "truth" that a person making an argument or assertion claims to know intuitively "from the gut" or because it "feels right" without regard to evidence, logic, intellectual examination, or facts.  
(Wikipedia, “Truthiness”)
- In Python, the following values are regarded as "false":  
**False**, **None**, any kind of zero, the empty string, any kind of empty sequence or empty container, and any user-defined object which defines a `__bool__()` method that returns a false value
- "True" values are anything else
- As a matter of style, I strongly prefer the use of actual boolean values to denote **True** or **False**



# any

- `any(iterable)` returns `True` if any element of the *iterable* is “truthy”
  - `>>> any([None, 0, '', False, []])`  
`False`
  - `>>> any([None, 0, '', False, [], -1])`  
`True`



# all

- `all(iterable)` returns `True` if every element of the *iterable* is “truthy”
  - ```
>>> all([1, 2 < 3, 4])  
True
```
  - ```
>>> all([1, 2 > 3, 4])  
False
```



# forall and some

- In my opinion, it would be more useful to have functions that apply a predicate to every element of a sequence
  - `forall(predicate, sequence)` should return `True` if all elements satisfy the *predicate*
  - `some(predicate, sequence)` should return `True` if any element of a sequence satisfies the *predicate*
- Since Python does not seem to provide these, I'll write them myself



# forall

- ```
>>> def forall(f, s):  
    return all(map(f, s))
```
- ```
>>> forall(lambda x: x > 0, [1, 2, 3])  
True
```
- ```
>>> forall(lambda x: x > 0, [1, -2, 3])  
False
```



# some

- ```
>>> def some(f, s):  
    return any(map(f, s))
```
- ```
>>> some(lambda x: x in 'aeiou', 'frog')
```

  
**True**
- ```
>>> some(lambda x: x in 'aeiou', 'fly')
```

  
**False**
- Higher-level functions are useful partly because they let you replace for loops with more concise, easier to understand function calls



# List comprehensions

- A *list comprehension* is a way of creating a list
- **Syntax:**  
[*expression generator condition generator condition ... generator condition*]  
where there is at least one *generator*
- The *conditions* are all optional
- A *generator* has the syntax **for** *variable in sequence*
- A *condition* has the syntax **if** *test*



# List comprehension examples

- ```
>> [x * x for x in range(1, 11)]  
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```
- ```
>>> [x.upper() for x in "hello, there!" if x not in "aeiou"]  
['H', 'L', 'L', ',', ' ', 'T', 'H', 'R', '!']
```
- ```
>>> [(x, y) for x in range(1, 6) for y in range(1, 6) if x != y]  
[(1, 2), (1, 3), (1, 4), (1, 5), (2, 1), (2, 3), (2, 4), (2, 5),  
(3, 1), (3, 2),  
(3, 4), (3, 5), (4, 1), (4, 2), (4, 3), (4, 5), (5, 1), (5, 2),  
(5, 3), (5, 4)]
```



# Conclusion

- Python is *not* a functional language; it is an object-oriented language with some functional features
- Those features are well worth learning
- The best current source is probably [Functional Programming HOWTO](#) by A. M. Kuchling



# The End

You think you know when you learn, are more sure when you can write, even more when you can teach, but certain when you can program.

-- Alan J. Perlis