# Programming Paradigms

<table>
<thead>
<tr>
<th>Paradigm</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Procedural</strong></td>
<td>Sequence of instructions that inform the computer what to do with the program's input</td>
<td>C, Pascal, Unix (sh)</td>
</tr>
<tr>
<td><strong>Object-Oriented</strong></td>
<td>Deal with collections of objects which maintain internal state and support methods that query or modify this internal state in some way.</td>
<td>Java, Smalltalk</td>
</tr>
<tr>
<td><strong>Declarative</strong></td>
<td>Specification describes the problem to be solved, and language implementation figures out the details</td>
<td>SQL, Prolog</td>
</tr>
<tr>
<td><strong>Functional</strong></td>
<td>Decomposes into a set of functions, each of which solely takes inputs and produces outputs with no internal state.</td>
<td>Haskell, OCaml, ML</td>
</tr>
</tbody>
</table>
# Programming Paradigms

<table>
<thead>
<tr>
<th>Paradigm</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Procedural</strong></td>
<td>Sequence of instructions that inform the computer what to do with the program's input</td>
<td>C, Pascal, Unix (sh)</td>
</tr>
<tr>
<td><strong>Object-Oriented</strong></td>
<td>Deal with collections of objects which maintain internal state and support methods that query or modify this internal state in some way.</td>
<td>Java, Smalltalk</td>
</tr>
<tr>
<td><strong>Declarative</strong></td>
<td>Specification describes the problem to be solved, and language implementation figures out the details</td>
<td>SQL, Prolog</td>
</tr>
<tr>
<td><strong>Functional</strong></td>
<td>Decomposes into a set of functions, each of which solely takes inputs and produces outputs with no internal state.</td>
<td>Scala, C++, Python</td>
</tr>
<tr>
<td><strong>Multi-Paradigm</strong></td>
<td>Supports several different paradigms, to be combined freely</td>
<td></td>
</tr>
</tbody>
</table>
Functional Programming Concepts
Functional Programming Concepts

Primary entity is a "function"
Functional Programming Concepts

Primary entity is a "function"

"Pure" functions are mathematical
Functional Programming Concepts

Primary entity is a "function"

"Pure" functions are mathematical
  Output depends only on input
Functional Programming Concepts

Primary entity is a "function"

"Pure" functions are mathematical
  Output depends only on input
  No side effects that modify internal state
Functional Programming Concepts

Primary entity is a "function"

"Pure" functions are mathematical
  Output depends only on input
  No side effects that modify internal state

print() and file.write() are side effects
Functional Programming Concepts

Primary entity is a "function"

"Pure" functions are mathematical
  
  Output depends only on input

  No side effects that modify internal state

  \texttt{print()} and \texttt{file.write()} are side effects

Strict (Haskell): no assignments, variables, or state
Functional Programming Concepts

Primary entity is a "function"

"Pure" functions are mathematical

Output depends only on input

No side effects that modify internal state

print() and file.write() are side effects

Strict (Haskell): no assignments, variables, or state

Flexible (Python): encourage low-interference functions
Functional Programming Concepts

Primary entity is a "function"

"Pure" functions are mathematical
  Output depends only on input
  No side effects that modify internal state

  `print()` and `file.write()` are side effects

Strict (Haskell): no assignments, variables, or state

Flexible (Python): encourage low-interference functions
  Functional-looking interface but use variables, state internally
Why Functional Programming?
Why Functional Programming?

Why avoid objects and side effects?
Why Functional Programming?

Why avoid objects and side effects?

Formal Provability Line-by-line invariants
Why Functional Programming?

Why avoid objects and side effects?

**Formal Provability** Line-by-line invariants

**Modularity** Encourages small independent functions
Why Functional Programming?

Why avoid objects and side effects?

**Formal Provability** Line-by-line invariants

**Modularity** Encourages small independent functions

**Composability** Arrange existing functions for new goals
Why Functional Programming?

Why avoid objects and side effects?

**Formal Provability** Line-by-line invariants

**Modularity** Encourages small independent functions

**Composability** Arrange existing functions for new goals

**Easy Debugging** Behavior depends only on input
Why Functional Programming?

Why avoid objects and side effects?

**Formal Provability** Line-by-line invariants

**Modularity** Encourages small independent functions

**Composability** Arrange existing functions for new goals

**Easy Debugging** Behavior depends only on input

Let's Get Started!
Common Pattern
output = []
for element in iterable:
    val = function(element)
    output.append(val)
return output
Common Pattern

```python
output = []
for element in iterable:
    val = function(element)
    output.append(val)
return output
```

```python
return [function(element) for element in iterable]
```
Common Pattern

```python
output = []
for element in iterable:
    val = function(element)
    output.append(val)
return output

return [function(element)
         for element in iterable]
```
output = []
for element in iterable:
    val = function(element)
    output.append(val)
return output

return [function(element) for element in iterable]
[len(s) for s in languages]
["python", "perl", "java", "c++"]

len
6

\[\text{len}()\text{ for } s \text{ in } \text{languages}\]
\[
["python", "perl", "java", "c++"]
\]
[len(s) for s in languages]

["python", "perl", "java", "c++"]

len

6
len(s) for s in languages

["python", "perl", "java", "c++"]

len

6

4
[len(s) for s in languages]

["python", "perl", "java", "c++"]

6 4

len
len(s) for s in languages
["python", "perl", "java", "c++"]

6 4 4
```python
[for s in languages:
    len(s)]
```

```python
"python", "perl", "java", "c++"
```

```
6
4
4
```
[len(s) for s in languages]
["python", "perl", "java", "c++"]

6
4
4
3
[len(s) for s in languages]
[
"python", "perl", "java", "c++"
]
len(s) for s in languages

["python", "perl", "java", "c++"]

[6, 4, 4, 3]
map(fn, iter)

map :: (a → b) × [a] → [b]

No discussion of elements!
map(len, languages)

< 6 , 4 , 4 , 3 >
Another Common Pattern
Another Common Pattern

```python
output = []
for element in iterable:
    if predicate(element):
        output.append(val)
return output
```
output = []
for element in iterable:
    if predicate(element):
        output.append(val)
return output

[element for element in iterable
    if predicate(element)]
Another Common Pattern

```python
output = []
for element in iterable:
    if predicate(element):
        output.append(val)
return output
```

```
[element for element in iterable
    if predicate(element)]
```
Another Common Pattern

```python
def my_function(iterable, predicate):
    output = []
    for element in iterable:
        if predicate(element):
            output.append(element)
    return output
```

```python
[element for element in iterable if predicate(element)]
```
[num for num in fibs if is_even(num)]

[1, 1, 2, 3, 5, 8, 13, 21, 34]
[num for num in fibs if is_even(num)]

[1, 1, 2, 3, 5, 8, 13, 21, 34]
[num for num in fibs if is_even(num)]

[1, 1, 2, 3, 5, 8, 13, 21, 34]
[num for num in fibs if is_even(num)]
[1, 1, 2, 3, 5, 8, 13, 21, 34]
[num for num in fibs if is_even(num)]

[1, 1, 2, 3, 5, 8, 13, 21, 34]
[num for num in fibs if is_even(num)]
[1, 1, 2, 3, 5, 8, 13, 21, 34]
\[\text{[num for num in fibs if is\_even(num)]}\]
\[1, 1, 2, 3, 5, 8, 13, 21, 34]\]
[num for num in fibs if is_even(num)]

[1, 1, 2, 3, 5, 8, 13, 21, 34]
[num for num in fibs if is_even(num)]

[1, 1, 2, 3, 5, 8, 13, 21, 34]
[num for num in fibs if is_even(num)]

[1, 1, 2, 3, 5, 8, 13, 21, 34]
[num for num in fibs if is_even(num)]

[1, 1, 2, 3, 5, 8, 13, 21, 34]
[num for num in fibs if is_even(num)]

[1, 1, 2, 3, 5, 8, 13, 21, 34]

Only keep elements that satisfy some predicate

[2, 8, 34]
filter :: (a -> bool) x [a] -> [a]
filter(is_even, fibs)

< 2, 8, 34 >

[num for num in fibs if is_even(num)]

[1, 1, 2, 3, 5, 8, 13, 21, 34]
Lambda Functions
Anonymous, on-the-fly, unnamed functions

Lambda Functions
Keyword `lambda` creates an anonymous function

```
lambda params: expr(params)
```

Returns an expression
Defined Functions vs. Lambdas
def greet():
    print("Hi!")

def binds a function object to a name

Defined Functions vs. Lambdas
defined Functions vs. Lambdas

```python
def greet():
    print("Hi!"))
lambda val: val ** 2
lambda x, y: x * y
lambda pair: pair[0] * pair[1]
```

**Defined Functions vs. Lambdas**

- `def` binds a function object to a name.
- `lambda` only creates a function object.
def greet():
    print("Hi!")

lambda val: val ** 2
lambda x, y: x * y
lambda pair: pair[0] * pair[1]

(lambda x: x > 3)(4) # => True
Using Lambdas
Using Lambdas

\[
\text{triple} = \lambda x: x \times 3 \quad \# \text{NEVER EVER DO THIS}
\]
Using Lambdas

triple = lambda x: x * 3  # NEVER EVER DO THIS

# Squares from 0**2 to 9**2
map(lambda val: val ** 2, range(10))
triple = lambda x: x * 3  # NEVER EVER DO THIS

# Squares from 0**2 to 9**2
map(lambda val: val ** 2, range(10))

# Tuples with positive second elements
filter(lambda pair: pair[1] > 0, [(4, 1), (3, -2), (8, 0)]

Using Lambdas
Iterators and Generators
Represent data stream, returned one element at a time
Iterators
Iterators

Iterators are objects, like (almost) everything in Python

Represent finite or infinite data streams
Iterators

Iterators are objects, like (almost) everything in Python
 Represent finite or infinite data streams

Use `next(iterator)` to yield successive values
 Raises `StopIteration` error upon termination
Iterators

Iterators are objects, like (almost) everything in Python
Represent finite or infinite data streams

Use `next(iterator)` to yield successive values

Raises `StopIteration` error upon termination

Use `iter(data)` to build an iterator for a data structure
Iterable
# Build an iterator over [1,2,3]

```python
it = iter([1,2,3])
```
# Build an iterator over [1,2,3]

```python
it = iter([1,2,3])
```

```python
next(it)  # => 1
```
# Build an iterator over [1,2,3]

```python
it = iter([1,2,3])

next(it)  # => 1
next(it)  # => 2
```
# Build an iterator over [1,2,3]

it = iter([1,2,3])

next(it)  # => 1
next(it)  # => 2
next(it)  # => 3
Iterable

# Build an iterator over [1,2,3]

it = iter([1,2,3])

next(it)  # => 1
next(it)  # => 2
next(it)  # => 3
next(it)  # raises StopIteration error
For Loops use Iterators
For Loops use Iterators

```python
for data in data_source:
    process(data)
```
For Loops use Iterators

```python
for data in data_source:
    process(data)

# is really
for data in iter(data_source):
    process(data)
```
For Loops use Iterators

```python
for data in data_source:
    process(data)

# is really
for data in iter(data_source):
    process(data)
```

**Iterator sees changes to the underlying data structure**
Builtins use Iterators
Builtins use Iterators

# Return a value

\texttt{max(iterable)}  \hspace{1cm} \texttt{min(iterable)}

\texttt{val \texttt{in} iterable}  \hspace{1cm} \texttt{val \texttt{not in} iterable}

\texttt{all(iterable)}  \hspace{1cm} \texttt{any(iterable)}
Builtins use Iterators

# Return a value
max(iterable)
val in iterable
all(iterable)

min(iterable)
val not in iterable
any(iterable)

Consume iterable until return value is known
Builtins use Iterators

# Return a value
max(iterable)
val in iterable
all(iterable)

min(iterable)
val not in iterable
any(iterable)

What happens for infinite iterators?
Builtins use Iterators

# Return a value
max(iterable)
val in iterable
all(iterable)

# Return an iterator
enumerate(iterable)
map(fn, iterable)

Consume iterable until return value is known
min(iterable)
val not in iterable
any(iterable)

What happens for infinite iterators?
zip(*iterables)
filter(pred, iterable)
Builtins use Iterators

# Return a value
max(iterable)
val in iterable
val not in iterable
all(iterable)
min(iterable)
any(iterable)

# Return an iterator
enumerate(iterable)
map(fn, iterable)
zip(*iterables)
filter(pred, iterable)

What happens for infinite iterators?

To convert to list, use list(iterable)
Generators
"Resumable Functions"

Generators
Regular Functions vs. Generator Functions
Regular Functions vs. Generator Functions

Regular Functions

Generators
## Regular Functions vs. Generator Functions

<table>
<thead>
<tr>
<th>Regular Functions</th>
<th>Generators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return a single, computed value</td>
<td>Return an iterator that generates a stream of values</td>
</tr>
</tbody>
</table>
# Regular Functions vs. Generator Functions

<table>
<thead>
<tr>
<th>Regular Functions</th>
<th>Generators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return a single, computed value</td>
<td>Return an iterator that \textit{generates} a stream of values</td>
</tr>
<tr>
<td>Each call generates a new private namespace and new local variables, then variables are thrown away</td>
<td>Local variables aren't thrown away when exiting a function – you can resume where you left off!</td>
</tr>
</tbody>
</table>
Simple Generator
def generate_ints(n):
    for i in range(n):
        yield i

Simple Generator

The `yield` keyword tells Python to convert the function into a generator.
def generate_ints(n):
    for i in range(n):
        yield i

g = generate_ints(3)

The `yield` keyword tells Python to convert the function into a generator.
def generate_ints(n):
    for i in range(n):
        yield i

g = generate_ints(3)
type(g)  # => <class 'generator'>

The `yield` keyword tells Python to convert the function into a generator.
def generate_ints(n):
    for i in range(n):
        yield i

g = generate_ints(3)
type(g)  # => <class 'generator'>
next(g)  # => 0
def generate_ints(n):
    for i in range(n):
        yield i

g = generate_ints(3)
type(g)  # => <class 'generator'>
next(g)  # => 0
next(g)  # => 1

The `yield` keyword tells Python to convert the function into a generator.
def generate_ints(n):
    for i in range(n):
        yield i

g = generate_ints(3)

print(type(g))  # => <class 'generator'>
print(next(g))  # => 0
print(next(g))  # => 1
print(next(g))  # => 2

The yield keyword tells Python to convert the function into a generator.
def generate_ints(n):
    for i in range(n):
        yield i

g = generate_ints(3)
type(g)    # => <class 'generator'>
next(g)    # => 0
next(g)    # => 1
next(g)    # => 2
next(g)    # raises StopIteration

The yield keyword tells Python to convert the function into a generator.
def generate_fibs():
a, b = 0, 1
while True:
    a, b = b, a + b
    yield a

Another Generator

Infinite data stream of Fibonacci numbers
Using Our Generator
Using Our Generator

g = generate_fibs()
Using Our Generator

g = generate_fibs()

next(g)  # => 1
Using Our Generator

g = generate_fibs()

next(g)  # => 1
next(g)  # => 1
Using Our Generator

g = generate_fibs()

next(g)  # => 1
next(g)  # => 1
next(g)  # => 2
next(g)  # => 3
next(g)  # => 5
g = generate_fibs()

next(g) # => 1
next(g) # => 1
next(g) # => 2
next(g) # => 3
next(g) # => 5
max(g)  # Oh no! What happens?
def fibs_under(n):
    for f in generate_fibs():  # Loops over 1, 1, 2, ...
        if f > n:
            break
    print(f)
Summary: Why Use Iterators and Generators?
Summary: Why Use Iterators and Generators?

Compute data on demand

Reduces in-memory buffering

Can avoid expensive function calls
Summary: Why Use Iterators and Generators?

Compute data on demand

- Reduces in-memory buffering
- Can avoid expensive function calls

`range, map, filter` and others are iterable
Summary: Why Use Iterators and Generators?

Compute data on demand

- Reduces in-memory buffering
- Can avoid expensive function calls

range, map, filter and others are iterable

Great for asynchronous programming (network/web)
Decorators
Functions as Arguments
Functions as Arguments

# map(fn, iterable)
# filter(pred, iterable)
Functions as Arguments

# map(fn, iterable)
# filter(pred, iterable)

def perform_twice(fn, *args, **kwargs):
    fn(*args, **kwargs)
    fn(*args, **kwargs)
    fn(*args, **kwargs)
Functions as Arguments

# map(fn, iterable)
# filter(pred, iterable)

def perform_twice(fn, *args, **kwargs):
    fn(*args, **kwargs)
    fn(*args, **kwargs)

perform_twice(print, 5, 10, sep='&', end='...')
# => 5&10...5&10...
Functions as Return Values
Functions as Return Values

def make_divisibility_test(n):
    def divisible_by_n(m):
        return m % n == 0
    return divisible_by_n
def make_divisibility_test(n):
    def divisible_by_n(m):
        return m % n == 0
    return divisible_by_n

div_by_3 = make_divisibility_test(3)
def make_divisibility_test(n):
    def divisible_by_n(m):
        return m % n == 0
    return divisible_by_n

div_by_3 = make_divisibility_test(3)
filter(div_by_3, range(10))  # generates 0, 3, 6, 9
Functions as Return Values

def make_divisibility_test(n):
    def divisible_by_n(m):
        return m % n == 0
    return divisible_by_n

div_by_3 = make_divisibility_test(3)
filter(div_by_3, range(10))  # generates 0, 3, 6, 9
make_divisibility_test(5)(10)  # => True
¿Por qué no los dos?

Functions as Arguments

\[ f(x) \]

\[ g(x') \]
¿Por qué no los dos?

x

\[ f \]

Functions as Arguments

f(x)

\[ g \]

Functions as Return Values

g(x')
¿Por qué no los dos?

Functions as Arguments

Decorator!

Functions as Return Values

f

f(x)

x

x'

g

g(x')
Writing Our First Decorator
Writing Our First Decorator

def debug(function):

def debug(function):
    def wrapper(*args, **kwargs):

Writing Our First Decorator
def debug(function):
    def wrapper(*args, **kwargs):
        print("Arguments:", args, kwargs)
Writing Our First Decorator

def debug(function):
    def wrapper(*args, **kwargs):
        print("Arguments:", args, kwargs)
        return function(*args, **kwargs)
def debug(function):
    def wrapper(*args, **kwargs):
        print("Arguments:", args, kwargs)
        return function(*args, **kwargs)
    return wrapper

Writing Our First Decorator
Using our debug decorator
Using our `debug` decorator

def foo(a, b, c=1):
    return (a + b) * c
Using our `debug` decorator

```python
def foo(a, b, c=1):
    return (a + b) * c

foo = debug(foo)
```
def foo(a, b, c=1):
    return (a + b) * c

foo = debug(foo)
foo(2, 3)  # prints "Arguments: (2, 3) {}"
# => returns 5
Using our `debug` decorator

```python
def foo(a, b, c=1):
    return (a + b) * c

foo = debug(foo)
foo(2, 3)  # prints "Arguments: (2, 3) {}"
# => returns 5
foo(2, 1, c=3)  # prints "Arguments: (2, 1) {'c': 3}"  
# => returns 9
```
Using our `debug` decorator

def foo(a, b, c=1):
    return (a + b) * c

foo = debug(foo)
foo(2, 3)  # prints "Arguments: (2, 3) {}"
# => returns 5
foo(2, 1, c=3)  # prints "Arguments: (2, 1) {'c': 3}"
# => returns 9
print(foo)  # <function debug.<locals>.wrapper at 0x...>
Using our `debug` decorator

```python
def foo(a, b, c=1):
    return (a + b) * c

foo = debug(foo)
foo(2, 3)  # prints "Arguments: (2, 3) {}"
# => returns 5
foo(2, 1, c=3)  # prints "Arguments: (2, 1) {'c': 3}"  
# => returns 9
print(foo)  # <function debug.<locals>.wrapper at 0x...>
```

It seems like overkill to say `foo` twice here.
Using our `debug` decorator
Using our `debug` decorator

```python
@debug
def foo(a, b, c=1):
    return (a + b) * c
```
Using our `debug` decorator

```python
@debug
def foo(a, b, c=1):
    return (a + b) * c
```
Using our `debug` decorator

```python
@debug
def foo(a, b, c=1):
    return (a + b) * c
```

@decorator applies a decorator to the following function
Using our `debug` decorator

```python
@debug
def foo(a, b, c=1):
    return (a + b) * c

foo(5, 3, c=2)  # prints "Arguments: (5, 3) {'c': 2}"
# => returns 16
```

@decorator applies a decorator to the following function
Credit

Python Documentation, of course

Guide to Functional Programming

A few other sites, which I've unfortunately forgotten.