Functional Programming

C-START Python PD Workshop

Functional Programming

- High-order functions

Functional Programming

- High-order functions
- We can do a lot in very few lines

Functional Programming

- High-order functions
- We can do a lot in very few lines
- Allow us to mathematically prove our algorithms correct, that's better than any finite amount of unit tests!

Functional Programming

- High-order functions
- We can do a lot in very few lines
- Allow us to mathematically prove our algorithms correct, that's better than any finite amount of unit tests!
- Decorators are a little piece of functional programming

Functional Programming

- High-order functions
- We can do a lot in very few lines
- Allow us to mathematically prove our algorithms correct, that's better than any finite amount of unit tests!
- Decorators are a little piece of functional programming
- Generator expressions are also a form of functional programming

Functions

Functions are *first-class citizens* in Python:

```
>>> def identity(x):
...     return x
...
>>> type(identity)
<class 'function'>
```

Functions

Functions are *first-class citizens* in Python:

```
>>> def identity(x):
...     return x
...
>>> type(identity)
<class 'function'>
```

Functions can also be written anonymously as `lambda`s:

```
>>> identity = lambda x:x
>>> identity(42)
42
```

Functions

Functions are *first-class citizens* in Python:

```
>>> def identity(x):
...     return x
...
>>> type(identity)
<class 'function'>
```

Functions can also be written anonymously as `lambda`s:

```
>>> identity = lambda x:x
>>> identity(42)
42
```

In this case, the first style is preferred. It's a bit easier to read, not to mention it's actually named.

`min/max`

`min/max` gets the minimum or maximum value from an iterable, optionally using a key function to select by.

min/max

`min/max` gets the minimum or maximum value from an iterable, optionally using a key function to select by.

Example:

```
x = min(points, key=lambda p:dist(p, z))
```

zip

`zip` creates a **iterator** over the nth element of each of it's arguments (which are iterables).

zip

`zip` creates a **iterator** over the nth element of each of it's arguments (which are iterables).

Example:

```
for a, b, c in zip(list1, list2, list3):
    # do something
```

zip

`zip` creates a **iterator** over the nth element of each of it's arguments (which are iterables).

Example:

```
for a, b, c in zip(list1, list2, list3):
    # do something
```

Pro Tip: Iterating over the columns of a 2D matrix

```
for col in zip(*M):
    # do something with each column
```

Other Functional Things

- `map(func, *iterables)`, which calls `func(*t)` for all `t` in `zip(*iterables)`. Note that `map` is completely unnecessary as the same can be done using generator expressions. Under a few cases, it may be better to use `map` to improve readability.

Other Functional Things

- `map(func, *iterables)`, which calls `func(*t)` for all `t` in `zip(*iterables)`. Note that `map` is completely unnecessary as the same can be done using generator expressions. Under a few cases, it may be better to use `map` to improve readability.
- `reduce(func, sequence)` which reduces a sequence by calling `func(func(func(a, b), c), ...)`. This is useful for taking the product of a sequence (use `operator.mul`)

Recommended Reading

The **Functional Programming HOWTO** page in the Python documentation has some very useful tips for functional programming.
`https://docs.python.org/howto/functional.html`