Generators & List Comprehensions

Iterable Functions

C-START Python PD Workshop

Generator Functions

Python provides a special kind of function which `yields` rather than `returns`. This **generator function** is effectively an efficient iterable. Consider the `range` function we have been using[1]:

```
def range(start, stop, step=1):
    i = 0
    while i < stop:
        yield i
        i += step
```

[1]This is actually a simplification

Generator Functions

Python provides a special kind of function which `yield`s rather than `return`s. This **generator function** is effectively an efficient iterable. Consider the `range` function we have been using[1]:

```python
def range(start, stop, step=1):
    i = 0
    while i < stop:
        yield i
        i += step
```

Generator functions are a certain kind of the more generic **generator**.

[1] This is actually a simplification

Generator Expressions

Generators can be written inline, these are called **generator expressions**.

```
(x + 4 for x in nums if x % 2 == 0)
```

Generator Expressions

Generators can be written inline, these are called **generator expressions**.

```
(x + 4 for x in nums if x % 2 == 0)
```

There's two parts to a generator expression:

1. Performing something for every element with `for...in`.

Generator Expressions

Generators can be written inline, these are called **generator expressions**.

```
(x + 4 for x in nums if x % 2 == 0)
```

There's two parts to a generator expression:

1. Performing something for every element with `for...in`.
2. Selecting a subset of elements to operate on with `if`. This part is optional.

Expression Syntax

```
(expression for expr in sequence1
            if condition1
            for expr2 in sequence2
            if condition2
            for expr3 in sequence3 ...
            if condition3
            for exprN in sequenceN
            if conditionN)
```

Notice the loops are evaluated outside-in.

Applications of Generator Expressions

- Summing ASCII values of a string

  ```
  sum(ord(c) for c in s)
  ```

  Note that the double-parentheses can be omitted.

Applications of Generator Expressions

- Summing ASCII values of a string

  ```
  sum(ord(c) for c in s)
  ```

  Note that the double-parentheses can be omitted.

- File readers

  ```
  reader = (float(line) for line in f)
  while processing_queue:
      process(next(reader))
  ```

Applications of Generator Expressions

- Summing ASCII values of a string

  ```
  sum(ord(c) for c in s)
  ```

  Note that the double-parentheses can be omitted.

- File readers

  ```
  reader = (float(line) for line in f)
  while processing_queue:
      process(next(reader))
  ```

- Hash Function pRNGs

  ```
  rng = (hashfunc(x)/MAXHASH for x in count())
  diceroll(next(rng))
  ```

Applications of Generator Expressions

- Summing ASCII values of a string

  ```
  sum(ord(c) for c in s)
  ```

  Note that the double-parentheses can be omitted.

- File readers

  ```
  reader = (float(line) for line in f)
  while processing_queue:
      process(next(reader))
  ```

- Hash Function pRNGs

  ```
  rng = (hashfunc(x)/MAXHASH for x in count())
  diceroll(next(rng))
  ```

- The possibilities are endless!

List Comprehensions

Building lists in a syntax like generator expressions can be done simply by using square brackets.

```
my_list = [x + 4 for x in nums if x % 2 == 0]
```

List Comprehensions

Building lists in a syntax like generator expressions can be done simply by using square brackets.

```
my_list = [x + 4 for x in nums if x % 2 == 0]
```

Non-comprehensive Alternative

A novice Pythonist might choose this instead:

```
my_list = []
for x in nums:
    if x % 2 == 0:
        my_list.append(x)
```

Why use a comprehension? It's easier to read and faster.

Generic Comprehensions

The same comprehension syntax can be applied to other data structures like so:

```
# Sets
myset = {foo(x, y) for x, y in points}

# Dictionaries
mydict = {point: dist(p) for p in points}

# Tuples
mytup = tuple(foo(x, y) for x, y in points)
```