Programming Paradigms

Lecture 4. List comprehension. Functional Python and JavaScript

Test N°3

See Moodle

Outline

- Generators recap
- Closures
- Mapping multiple lists
- List comprehension
- Functional programming in Python
- Functional programming in JavaScript

Clarification: generators

```
(define (loop students)
  (cond
    [(empty? students)
     "no more 4.0+ students"]
    [(> (cdr (first students)) 4.0)
     (begin
       (yield (car (first
students)))
       (loop (rest students)))]
    [else (loop (rest students))]))

(define students-with-4+
  (generator () (loop students)))
```

```
(students-with-4+) ; "Anna"
(students-with-4+) ; "Charles"
(students-with-4+) ; "no more 4.0+ students"
```

Closures

```
(define (less-than n)
  (lambda (x) (< x n)))
```

Closures

```
(define (less-than n)
  (lambda (x) (< x n)))
```

Value of **n** is captured and stored together with the returned function.

Closures

```
(define (less-than n)
  (lambda (x) (< x n)))
```

Value of **n** is captured and stored together with the returned function.

In fact, we return a **closure.**
A **closure** is a function together with its environment (values of captured variables).

Closures

```
(define (less-than n)
  (lambda (x) (< x n)))

(filter (less-than 5) '(1 9 2 8 4 3 7))
; '(1 2 4 3)
```

Closures

```
(define (less-than n)
  (lambda (x) (< x n)))

(filter (less-than 5) '(1 9 2 8 4 3 7))
; '(1 2 4 3)

(define (satisfies-all? x predicates)
  (andmap (lambda (p) (p x)) predicates))
```

Closures

```
(define (less-than n)
  (lambda (x) (< x n)))

(filter (less-than 5) '(1 9 2 8 4 3 7))
; '(1 2 4 3)

(define (satisfies-all? x predicates)
  (andmap (lambda (p) (p x)) predicates))
```

Value of **x** is captured here.

Closures

```
(define (less-than n)
  (lambda (x) (< x n)))

(filter (less-than 5) '(1 9 2 8 4 3 7))
; '(1 2 4 3)

(define (satisfies-all? x predicates)
  (andmap (lambda (p) (p x)) predicates))

(satisfies-all? 5 (list (less-than 8) (less-than 5)))
; #f
```

Closures

```
(define (less-than n)
  (lambda (x) (< x n)))

(filter (less-than 5) '(1 9 2 8 4 3 7))
; '(1 2 4 3)

(define (satisfies-all? x predicates)
  (andmap (lambda (p) (p x)) predicates))

(satisfies-all? 5 (list (less-than 8) (less-than 5)))
; #f

(satisfies-all? 5 (map less-than '(7 9 8 1 6)))
; #f
```

List comprehension

```
(for/list ([i '(1 2)]
           [j '("a" "b")])
  (cons i j))
; '((1 . "a") (2 . "b"))
```

`for/list` traverses lists in parallel

List comprehension

```
(for/list ([i '(1 2)]
           [j '("a" "b")])
  (cons i j))
; '((1 . "a") (2 . "b"))
```

`for/list` traverses lists in parallel

```
(for*/list ([i '(1 2)]
            [j '("a" "b")])
  (cons i j))
; '((1 . "a") (1 . "b") (2 . "a") (2 . "b"))
```

`for*/list` traverses lists independently (similarly to nested **`for`** loops)

List comprehension

```
(for/list ([i '(1 2)]
           [j '("a" "b")])
  (cons i j))
; '((1 . "a") (2 . "b"))
```

for/list traverses lists in parallel

```
(for*/list ([i '(1 2)]
            [j '("a" "b")])
  (cons i j))
; '((1 . "a") (1 . "b") (2 . "a") (2 . "b"))
```

for*/list traverses lists independently (similarly to nested **for** loops)

```
(for*/list ([i '(1 2)]
            [j '(1 2)]
            #:when (<= i j))
  (cons i j))
; '((1 . 1) (1 . 2) (2 . 2))
```

#:when and **#:unless** can be used to filter some combinations

Traits of functional programming: immutability

```
(define x 2)
(define x 3)
```

```
module: identifier already defined in: x
```

By default in functional languages data (values) are **immutable**!

Traits of functional programming: immutability

```
(define x 2)
(define x 3)

module: identifier already defined in: x
```

By default in functional languages data (values) are **immutable**!

```
(define numbers '(2 3 4))
(define more-numbers (cons 1 numbers))
```

One of the benefits is cheap copying — we only copy reference!

Traits of functional programming: referential transparency

```
(define (square x) (* x x))

(define (sum-of-squares x y)
  (+ (square x) (square y)))

(sum-of-squares 3 4)
```

Traits of functional programming: referential transparency

```
(define (square x) (* x x))

(define (sum-of-squares x y)
  (+ (square x) (square y)))

(sum-of-squares 3 4)

(+ (square 3) (square 4))
```

Traits of functional programming: referential transparency

```
(define (square x) (* x x))

(define (sum-of-squares x y)
  (+ (square x) (square y)))

(sum-of-squares 3 4)

(+ (square 3) (square 4))

(+ (* 3 3) (* 4 4))
```

Python: iterators

```
>>> lst = [1, 2, 3]
>>> it = iter(lst)
>>> it
<listiterator object at 0x10fc645d0>
```

iter attempts to convert an object into an iterator

Python: iterators

```
>>> lst = [1, 2, 3]
>>> it = iter(lst)
>>> it
<listiterator object at 0x10fc645d0>
>>> next(it)
1
```

next returns the next value of the iterator

Python: iterators

```
>>> lst = [1, 2, 3]
>>> it = iter(lst)
>>> it
<listiterator object at 0x10fc645d0>
>>> next(it)
1
>>> next(it)
2
```

next returns the next value of the iterator

Python: iterators

```
>>> lst = [1, 2, 3]
>>> it = iter(lst)
>>> it
<listiterator object at 0x10fc645d0>
>>> next(it)
1
>>> next(it)
2
>>> next(it)
3
```

next returns the next value of the iterator

Python: iterators

```
>>> lst = [1, 2, 3]
>>> it = iter(lst)
>>> it
<listiterator object at 0x10fc645d0>
>>> next(it)
1
>>> next(it)
2
>>> next(it)
3
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Python: iterators

```
>>> lst = [1, 2, 3]
>>> it = iter(lst)
>>> it
<listiterator object at 0x10fc645d0>
>>> next(it)
1
>>> next(it)
2
>>> next(it)
3
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

```
for i in iter(lst):
    print(i)
```

is the same as

```
for i in lst:
    print(i)
```

Python: generators and list comprehension

```
>>> squares_iter = (x*x for x in [1, 2, 3, 4, 5])
```

Python: generators and list comprehension

```
>>> squares_iter = (x*x for x in [1, 2, 3, 4, 5])
>>> next(squares_iter)
1
>>> next(squares_iter)
4
>>> next(squares_iter)
9
```

Python: generators and list comprehension

```
>>> squares_iter = (x*x for x in [1, 2, 3, 4, 5])
>>> next(squares_iter)
1
>>> next(squares_iter)
4
>>> next(squares_iter)
9

>>> squares_lst = [x*x for x in [1, 2, 3, 4, 5]]
>>> squares_lst
[1, 4, 9, 16, 25]
```

Python: generators

```
def nats():
    x = 0
    while True:
        yield x
        x += 1
```

Python: generators

```
def nats():
    x = 0
    while True:
        yield x
        x += 1

>>> nat = nats()
```

Python: generators

```
def nats():
    x = 0
    while True:
        yield x
        x += 1

>>> nat = nats()
>>> next(nat)
0
>>> next(nat)
1
>>> next(nat)
2
```

Python: generators

```
def nats_to(n):
    x = 0
    while x <= n:
        yield x
        x += 1

>>> [x for x in nats_to(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Python: map and filter

```
def square(x):
  return x*x

>>> map(square, nats_to(5))
[0, 1, 4, 9, 16, 25]

>>> map(lambda x: x*x, nats_to(5))
[0, 1, 4, 9, 16, 25]

>>> filter(lambda x: x % 2 == 0, nats_to(10))
[0, 2, 4, 6, 8, 10]
```

Python: functools module

```
>>> import functools
>>> functools.reduce(lambda c, x: c * (x + 1), nats_to(4), 1)
120
```

Python: closures and variables

```
def less_than(n):
    def f(x):
        return x < n
    return f

predicates = [None] * 10
for n in range(10):
    predicates[n] = less_than(n)

predicates[1](5) # what is the result?
```

Python: closures and variables

```
def less_than(n):
    def f(x):
        return x < n
    return f

predicates = [None] * 10
for n in range(10):
    predicates[n] = less_than(n)

predicates[1](5) # False – since 5 is not less than 1
```

Python: closures and variables

```
def less_than(n):
    return (lambda x: x < n)

predicates = [None] * 10
for n in range(10):
    predicates[n] = less_than(n)

predicates[1](5) # what is the result?
```

Python: closures and variables

```
def less_than(n):
    return (lambda x: x < n)

predicates = [None] * 10
for n in range(10):
    predicates[n] = (lambda x: x < n)

predicates[1](5) # what is the result?
```

Python: closures and variables

```
def less_than(n):
    return (lambda x: x < n)

predicates = [None] * 10
for n in range(10):
    predicates[n] = (lambda x: x < n)

predicates[1](5) # what is the result?
```

Variable **n** is captured in this closure!

Python: closures and variables

```
def less_than(n):
    return (lambda x: x < n)

predicates = [None] * 10
for n in range(10):
    predicates[n] = (lambda x: x < n)

predicates[1](5) # True — because n is a mutable variable
                 # and when we evaluate this, we have n = 9
```

Variable **n** is captured in this closure!
We capture **reference**, not value!

Python: closures and variables

```
def less_than(n):
    return (lambda x: x < n)

predicates = [None] * 10
for n in range(10):
    predicates[n] = (lambda x: x < n)

predicates[1](5) # True – because n is a mutable variable
                 # and when we evaluate this, we have n = 9
n = 4
predicates[1](5) # now this will be False since not (5 < 4)
```

Variable **n** is captured in this closure!
We capture **reference**, not value!

Python: closures and variables

```
def less_than(n):
    return (lambda x: x < n)

predicates = [None] * 10
for n in range(10):
    predicates[n] = (lambda x: x < n)

predicates[1](5) # True – because n is a mutable variable
                 # and when we evaluate this, we have n = 9
n = 4
predicates[1](5) # now this will be False since not (5 < 4)
```

Python: closures and variables

```
def less_than(n):
    return (lambda x: x < n)

predicates = [None] * 10
for n in range(10):
    predicates[n] = (lambda x, n=n: x < n)

predicates[1](5) # False
```

Add parameter with the same name and default value.

Python: closures and variables

```
def less_than(n):
    return (lambda x: x < n)

predicates = [None] * 10
for n in range(10):
    predicates[n] = (lambda x, n=n: x < n)

predicates[1](5) # False
```

Pass **value** of variable n as default, so that it is preserved in the closure.

Add parameter with the same name and default value.

Python: closures and variables

```
predicates = [(lambda x: x < n) for n in range(10)]

predicates[1](5) # what is the result?
```

Python: closures and variables

```
predicates = [(lambda x: x < n) for n in range(10)]

predicates[1](5) # True — because n is again a mutable variable
                 # and when we evaluate this, we have n = 9
```

Python: closures and variables

Add parameter with the same name and default value.

```
predicates = [(lambda x, n=n: x < n) for n in range(10)]

predicates[1](5) # False
```

JavaScript: closures and variables

```
function less_than(n) {
  return function(x) { return x < n };
}

predicates = [];
for (var n = 0; n < 10; n++) {
  predicates.push(less_than(n));
}
predicates[1](5); // what is the result?
```

JavaScript: closures and variables

```
function less_than(n) {
  return function(x) { return x < n };
}

predicates = [];
for (var n = 0; n < 10; n++) {
  predicates.push(less_than(n));
}
predicates[1](5); // false — since 5 is not less than 1
```

JavaScript: closures and variables

```
function less_than(n) {
  return function(x) { return x < n };
}

predicates = [];
for (var n = 0; n < 10; n++) {
  predicates.push(function(x) { return x < n; });
}
predicates[1](5); // what is the result?
```

JavaScript: closures and variables

```
function less_than(n) {
  return function(x) { return x < n };
}

predicates = [];
for (var n = 0; n < 10; n++) {
  predicates.push(function(x) { return x < n; });
}
predicates[1](5); // true – because n is a mutable variable
```

Variable **n** is captured in this closure!
We capture **reference**, not value!

JavaScript: closures and variables

```
function less_than(n) {
  return function(x) { return x < n };
}

predicates = [];
for (var n = 0; n < 10; n++) {
  predicates.push(function(x, n=n) { return x < n; });
}
predicates[1](5);
```

JavaScript: closures and variables

```
function less_than(n) {
  return function(x) { return x < n };
}

predicates = [];
for (var n = 0; n < 10; n++) {
  predicates.push(function(x, n=n) { return x < n; });
}
predicates[1](5);

ReferenceError: Cannot access uninitialized variable.
```

JavaScript: closures and variables

```
function less_than(n) {
  return function(x) { return x < n };
}

predicates = [];
for (var n = 0; n < 10; n++) {
  predicates.push(function(n){
    return function(x) { return x < n; }
  }(n));
}
predicates[1](5);
```

What what the most unclear part of the lecture for you?

See Moodle

References

1. Python Docs — Functional Programming HOWTO
2. Brendan Eich’s blog post on the history of JavaScript
3.