Lecture 5. Closures. Functional programming in Python and JavaScript

---

Nikolai Kudasov

Programming Paradigms (Fall 2025)

Innopolis, Russia, Sep 23, 2025

Lab of Programming Languages and Compilers

INNOPOLIS UNIVERSITY

Quiz for Lecture 4

Outline

1. Purely Functional Programming
2. Closures
3. Functional Programming in Python
4. Functional Programming in JavaScript

Purely Functional Programming

Purely Functional Programming Concepts

We have seen the following concepts of purely functional programming in Racket:

- pure functions
- immutable data structures
- tail recursion and local state
- higher-order functions
- list comprehensions

Let us briefly revisit these before comparing with Python and JavaScript.

Pure Functions

Pure functions are functions whose output depends purely on its input, i.e. it cannot read or write any global (mutable) state, disrupt control flow of a program (e.g. exceptions), involve non-determinism (e.g. random number generators), or perform I/O (e.g. read user input).

```
(define (map f l)
  (cond [(empty? l) empty]
        [else (cons (f (first l))
                    (map f (rest l)))]))
```

Pros:

+ equational reasoning
+ easier concurrency and parallelism

Cons:

− expensive random-access modification
− explicit effects (e.g. local state via the accumulator parameter)

Immutability

With purely-functional style, all variables and data are immutable, meaning that a value cannot change in the course of a program execution.

```
(define x 2)
(define x 3) ; error: identifier already defined in: x
```

Pros:

+ cheap copying of data structures
+ equational reasoning
+ easier concurrency (e.g. STM)
+ easier non-strict evaluation strategies

Cons:

− expensive random-access modification
− explicit local state (via the accumulator parameter)

Tail Recursion and Local State

Functional programming replaces the `for`-loops with recursive functions. To avoid stack overflow, it is often important to rely on **tail recursion**[1]. Tail recursion is often accompanied by the **accumulator parameter** pattern to model local state:

```
(define (my-sum lst)
  (define (helper lst current)
    (cond [(empty? lst) current]
          [else (helper (rest lst) (+ current (first lst)))]))
  (helper lst 0))
```

Pros:

+ preserves the pros of purely-functional programming

Cons:

− syntactically noisy

− less convenient to refactor

[1] This, however, is mostly a problem with strict languages. Lazy languages employ a different mechanism for function calls, so tail recursion is not as important there.

Higher-Order Functions

Higher-order functions allow parametrizing functions (algorithms) with other functions (algorithms).

```
(map car
     (filter (lambda (student) (> (cdr student) 4.0))
             students))
```

Pros:

+ succinct code
  (can say a lot with a few functions)
+ equational reasoning
+ abstracting over implementation
+ user-defined control structures[a]

Cons:

– dense code (a lot to unpack)
– efficient only with an optimizing compiler

[a] With special forms or lazy evaluation.

List Comprehensions

List comprehensions simplify iterating over and generating lists.

```
(for/list ([student students]
           #:when (> (cdr student) 4.0))
  (car student))
```

Pros:

+ short and readable code
+ pros of purely functional style

Cons:

− many flavors to select from

Closures

Closures by example (1 of 2)

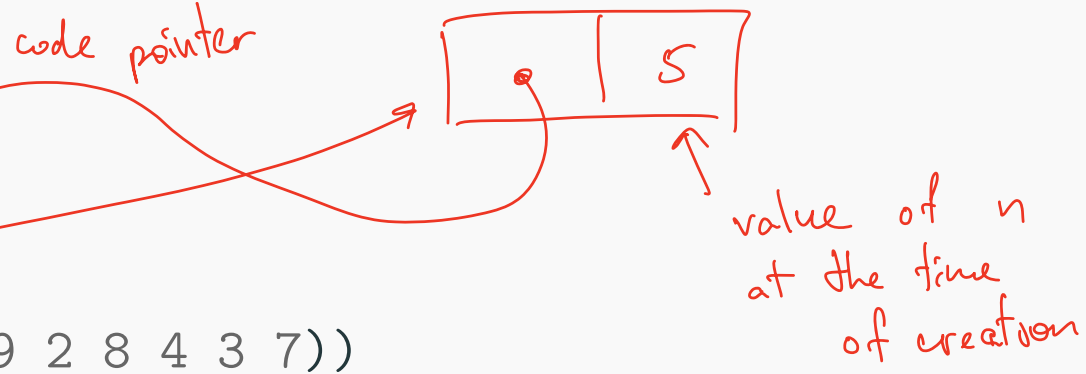
Consider this function:

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

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

Closures by example (1 of 2)

Consider this function:

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

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



1. Value of `n` is captured and has to be stored somehow in the returned function value.

Closures by example (1 of 2)

Consider this function:

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

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

1. Value of `n` is captured and has to be stored somehow in the returned function value.
2. In fact, `less-than` returns a **closure**.

Closures by example (1 of 2)

Consider this function:

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

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

1. Value of `n` is captured and has to be stored somehow in the returned function value.
2. In fact, `less-than` returns a **closure**.

A **closure** is a function (code) together with its environment (values of free variables).

(pointer)

Closures by example (2 of 2)

Closures are often used with higher-order functions:

```
(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
```

Here `x` is captured in the `lambda`-expression.

5 min break

Functional Programming in Python

Purely Functional Programming in Python

Python has limited support for functional programming:

- purity of functions is maintained by the programmer
- standard data structures in Python are mutable
- tail recursion and local state can be maintained similarly to Racket
- standard higher-order functions exist and others can be defined by the programmer
- list comprehensions exist, but with some caveats w.r.t. closure and mutable variables

Iterators in Python

Python relies on (imperative) iterators in many places:

```
>>> 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
```

Iterators in Loops in Python

In particular, iterators are automatically used in `for`-loops:

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

is the same as

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

Generators in Python

Generators in Python are produced similarly to Racket, via `yield`:

```
def squares(list):
  for x in list:
    yield x*x

squares1 = squares([1, 2, 3])          # creating a generator
squares2 = (x*x for x in [1, 2, 3])    # generator comprehension

squares3 = list(squares([1, 2, 3]))    # list of results of a generator
squares4 = [x*x for x in [1, 2, 3]]    # list comprehension
```

Generators in Python (natural numbers)

Generators in Python are produced similarly to Racket, via `yield`:

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

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

Generators in Python (range)

Generators may produce a range to iterate over:

```
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]
```

Generators and Comprehensions in Python

Generators can be implicit (via a comprehension) and converted to lists:

```
def squares(list):
  for x in list:
    yield x*x

squares1 = squares([1, 2, 3])          # creating a generator
squares2 = (x*x for x in [1, 2, 3])    # generator comprehension

squares3 = list(squares([1, 2, 3]))    # list of results of a generator
squares4 = [x*x for x in [1, 2, 3]]    # list comprehension
```

Higher-Order Functions over Generators in Python

Higher-order functions (e.g. `map` and `filter`) can work with generators:

```
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]
```

The `functools` Module in Python

More higher-order functions are available in the `functools` module:

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

Closures and Variables in Python (1 of 7)

Predict the output in the following Python program:

```
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?
```

Closures and Variables in Python (1 of 7)

Predict the output in the following Python program:

```
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?
```

Here, the result is `False`, since $5 \not< 1$!

Closures and Variables in Python (2 of 7)

Predict the output in the following Python program:

```
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?
```

Closures and Variables in Python (2 of 7)

Predict the output in the following Python program:

```
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?
```

Here, the result is `False`, since $5 \not< 1$!

Closures and Variables in Python (3 of 7)

Predict the output in the following Python program:

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

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

Closures and Variables in Python (3 of 7)

Predict the output in the following Python program:

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

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

Here, the result is `True`, because the closure captures a **mutable** reference, not the numeric value of `n`!

Closures and Variables in Python (4 of 7)

```
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)
```

Closures and Variables in Python (5 of 7)

To fix the code, we need to capture the value of `n`. We can do this by adding an extra lambda argument with a default value:

```
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
```

Closures and Variables in Python (6 of 7)

Predict the output in the following Python program:

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

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

Closures and Variables in Python (6 of 7)

Predict the output in the following Python program:

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

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

Here, the result is `True`, because the closure captures a **mutable** reference, not the numeric value of `n`!

Closures and Variables in Python (7 of 7)

Again, to fix the code, we need to capture the value of `n`. We can do this by adding an extra lambda argument with a default value:

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

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

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

Functional Programming in JavaScript

Purely Functional Programming in JavaScript

JavaScript has limited support for functional programming:

- historically, JavaScript is influenced a lot by LISP[2]
- purity of functions is maintained by the programmer
- standard data structures in JavaScript are mutable
- tail recursion and local state can be maintained similarly to Racket
- standard higher-order functions exist and others can be defined by the programmer
- array comprehensions were removed(!) from the standard
- there are still some caveats w.r.t. closures and mutable variables

[2] See Brendan Eich's blog post: https://brendaneich.com/2008/04/popularity/.

Closures and Variables in JavaScript (1 of 3)

Predict the output in the following JavaScript program:

```
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?
```

Closures and Variables in JavaScript (1 of 3)

Predict the output in the following JavaScript program:

```
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?
```

Here, the result is `False`, since $5 \not< 1$!

Closures and Variables in JavaScript (2 of 3)

Predict the output in the following JavaScript program:

```
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?
```

Closures and Variables in JavaScript (2 of 3)

Predict the output in the following JavaScript program:

```
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?
```

Here, the result is `True`, because the closure captures a **mutable** reference, not the numeric value of `n`!

Closures and Variables in JavaScript (3 of 3)

Again, to fix the code, we need to capture the value of `n`. We can do this by adding an extra lambda argument with a default value. Unfortunately, JavaScript does not support default values, so we need a more elaborate structure:

```
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);
```

Quick recap

1. A closure is a

2. Python's list comprehensions rely on
3. Closures and mutable references have to be

Quick recap

1. A closure is a function (code) together with its environment (values of free variables)
2. Python's list comprehensions rely on
3. Closures and mutable references have to be

Quick recap

1. A closure is a function (code) together with its environment (values of free variables)
2. Python's list comprehensions rely on mutable iterators
3. Closures and mutable references have to be

Quick recap

1. A closure is a function (code) together with its environment (values of free variables)
2. Python's list comprehensions rely on mutable iterators
3. Closures and mutable references have to be handled with care

What's Next

Today, in the labs you will

1. practice more with iterations, streams, and generators in Racket

Next week, we restart with purely functional programming in Haskell. To prepare:

1. Install Haskell (see `https://www.haskell.org/downloads/`)
2. Read *"Learn you a Haskell for Great Good"* §2, §4, §5 (see `http://learnyouahaskell.com/chapters`)
3. Implement a program that renders a Koch snowflake of a given rank in Haskell on Code.World platform (`https://code.world/haskell`)

Thank you!