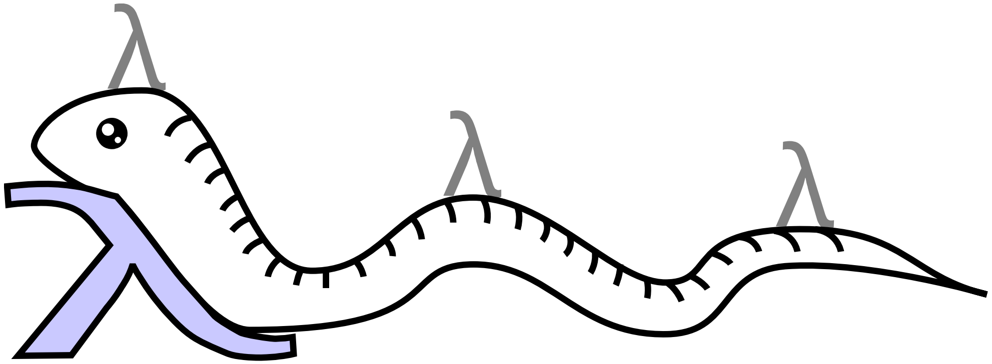
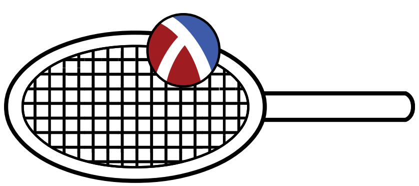
Python 3 for scientific computing ☆

Lecture 11, 25.4.2018
Functional programming (FP), part 2

Juha Jeronen
juha.jeronen@tut.fi



☆ *With a flavoring of Racket.*



Spring 2018, TUT, Tampere
RAK-19006 Various Topics of Civil Engineering

TAMPERE UNIVERSITY OF TECHNOLOGY

Meta

- The grand finale!

- Last week, we introduced Lisp and Racket, *so that we can now observe certain ideas in their natural habitat*:

  - Syntactic macros
  - Continuations
  - Pattern matching ☆
  - Currying ☆
  - High-level implementation of some programming language constructs

☆ To be fair, Haskell is *the* natural habitat of these two, but we had to pick just one language. In Racket, consider them as introduced species.

- **Why**? Even if you never actually program in Racket:
  - **Exposure** – encounter concepts from outside the Python community.
    - Python is much more advanced than C or Fortran, but there are ideas that could make your code shorter that Python doesn't currently employ. Part of the general knowledge of a software developer is to know them.
  - **Curiosity** – look under the hood of programming languages.
    - Understand the fundamentals; *see how code reduces to λ-calculus*. ☆ (Similar topics in mathematics: the construction of naturals and reals.)
  - **Techniques** – **get ideas to borrow into Python**.
    - We'll look at libraries that implement some of this stuff in Python.

☆ Exaggerating slightly. But not much.

Meta

Selective recap

- ***Primitives***, means of ***combination***, means of ***abstraction***
  –SICP, 1.1 The Elements of Programming

- Pure vs. impure FP

Here's an intro to pure FP
for the adventurous.

- First-class functions
- Higher-order functions
- Closures
- Partial application, currying

- **map**, **filter**, **fold**, **unfold**

- Basic syntax of Lisps, its distinctive features

- *Lambdas as code blocks* – code “more fluid than solid”
  - (A highly viscous fluid, obviously; but perhaps not this viscous.)

Macros

- *Scheme code is not meant to be written by humans, [but] … automatically by macros.*
  –Michele Simionato

- **What**: **Syntactic macros** transform the AST, by running arbitrary code on it, at compile time.
  - Contrast C preprocessor macros, which perform only text substitution.
  - Contrast C++ generics: *Lisp itself as metalanguage* (no separate templating mini-language).
  - StackOverflow. Understanding macros. Perl perspective. Macros and washing machines.
  - Paul Graham (1993): Programming bottom-up; Metaprogramming; Extensible programming.
  - *Lisp isn't a language, it's a building material. –Alan Kay* (of Smalltalk fame; on Lisp, [1] [2])

- **Why**: *Design patterns: a symptom of being unable to extract an abstraction* (Paul Graham).
  - E.g. **with** [1] or **assert** [2] in Python, which encode particular design patterns.
  - Syntactic macros allow *the programmer* to create such constructs:
    - **with** in Clojure.
    - Delayed evaluation in Racket.
  - Just like in mathematics [1] [2]: code is for humans, so ***notation matters***.
  - Macros are another important feature that make **Lisp more like a fluid than a solid**.
  - Democratization of language design? On the other hand, herd of cats (according to some, with machine guns), no BDFL. No process to pick, polish and promote the best abstractions; thus, "lowest common denominator" often used. *The Lisp Curse* [1] [2].

- History, in 30 seconds:
  - Early Lisp macros had the problems of *identifier capture* and *free symbol capture* [1] [2].
  - Scheme introduced *hygienic macros* [1] [2] specifically to solve those issues.
  - Racket adds a tower of phase levels; see short explanation and the paper by Flatt (2002).

- Syntactic macros are *almost* unique to the Lisp family. Exceptions: Julia, R.

Macros

When and how?

- **The nuclear option**: only create a macro if the job is not suitable for a run-of-the-mill function!
  - *Extract design patterns that cannot be extracted as functions*.
    - Although Racket is eager, macro arguments avoid immediate evaluation; highly useful [1][2].
    - ***Macros replacing design patterns*** is what “programs writing programs” means in Lisp; it's not about “source code generation” à la Cython (which takes Cython and writes C).
      - It's also robust, unlike source filters in many languages.
  - Add syntactic forms the original designer of the language might not approve. [1]
  - Create a ***DSL*** (*domain-specific language*) to fit the language to your domain; shorter code.
  - DRY out repetition in a set of similar macros, by *macro-writing macros*.
  - Programmatically create lookup tables at compile time. [1]

- **Limitations**:
  - **Second-class**; e.g. cannot compose on a macro, since it is expanded at compile time!
  - Local: a macro call cannot rewrite any forms *surrounding* it (due to Lisp's prefix notation)
  - Macros cannot change the lexical conventions (use of parentheses, prefix notation, …)
    - If you want to do that, you could modify or extend the ***reader*** [1] [2].

- In Racket, there are many macro definition tools; **the flagship is syntax-parse**.
  - syntax-parser and define-syntax-parser are just convenience forms on top of syntax-parse.
  - syntax-rules and syntax-case come from Scheme; simpler, but not quite as full-featured.
  - Start from Fear of Macros to learn the concepts, starting with the ***syntax transformer***. Then look at [1] [2] [3], examples on GitHub, and any examples you can dig up online (e.g. [1]).
  - Reference for syntax patterns and syntax classes; and directives such as #:fail-unless.

- Non-deterministic evaluation, automatic currying, Python-inspired syntactic forms, simple infix, User-programmable infix operators, Algebraic Data Types (ADTs) in Typed Racket.

Macros

A simple **for** loop

- Recall the functional looping strategy (lecture 3, slide 24). In Racket, we could write a looped hello world like this:

```
define looped-hello(n)
  define last {n - 1}
  let loop ([i 0])
    displayln
      format
        "hello from loop, iteration ~a"
        i
    cond
      {i < last}
        loop {i + 1}

(looped-hello 5)
```

The loop body.

- In the Scheme family, ***tail call optimization*** is **required** by the language spec, so this won't overflow the call stack.

- But we would have to repeat the marked lines whenever we wanted to make a loop.

- To turn this into a template, to make it reusable without having to remember a design pattern, we can define a macro:

```
require syntax/parse/define

define-syntax-parser for-range
  [_ (counter start end) body ...]
    syntax
      let ([last {end - 1}])
        let loop ([counter start])
          body
          ...
          cond
            {counter < last}
              loop {counter + 1}

for-range (i 0 5)
    displayln
      format
        "hello from loop, iteration ~a"
        i
```

Downloadable source file.

Macros

A simple **for** loop

- Recall the functional looping strategy (lecture 3, slide 24). In Racket, we could write a looped hello world like this:

```
define looped-hello(n)
  define last {n - 1}
  let loop ([i 0])
    displayln
      format
        "hello from loop, iteration ~a"
        i
    cond
      {i < last}
        loop {i + 1}

(looped-hello 5)
```

named let; "loop" is just a name.

The loop body.

- In the Scheme family, ***tail call optimization*** is **required** by the language spec, so this won't overflow the call stack.

- But we would have to repeat the marked lines whenever we wanted to make a loop.

- To turn this into a template, to make it reusable without having to remember a design pattern, we can define a macro:

```
require syntax/parse/define

define-syntax-parser for-range
  [_ (counter start end) body ...]
  syntax
    let ([last {end - 1}])
      let loop ([counter start])
        body
        ...
        cond
          {counter < last}
            loop {counter + 1}

for-range (i 0 5)
  displayln
    format
      "hello from loop, iteration ~a"
      i
```

_ means ignore. The first element is always the name of the macro itself, e.g. here "for-range".

Yes, "..."; syntax-parse uses pattern matching.

Downloadable source file.

Macros
Named **let**

Here we mean just "there may be more of these", although that's not too far from a syntax-parse pattern.

- We used **named let**, which is a looping and recursion construct:

```
define looped-hello(n)
  define last {n - 1}
  let loop ([i 0])
    displayln
      format
        "hello from loop, iteration ~a"
        i
    cond
      {i < last}
        loop {i + 1}

(looped-hello 5)
```

☆ To maintain backwards compatibility with traditional s-exprs, sweet-exp switches off indentation processing inside (), [] and {}, so there one must use traditional parenthesization.

For constructs like the bindings block of **let**, this is unfortunate. Another option is to use \\ (a.k.a. GROUP), also from SRFI-110, but it is a matter of opinion which looks more readable.

- Roughly, its usage is:

```
let loop-name ([var init] ...)
     body
     ...
     cond
       (loop-again?)
         loop-name next-value ...
```

Perform whatever check you want.

which expands to

```
letrec ([loop-name
          (λ (var …) body …
            (cond
              [(loop-again?)
               (loop-name next-value ...)]))])
     loop-name init ...
```

☆ *Should* be balanced …maybe?

- **The named let is also a macro!**
- This is what we meant when we said the Racket language is mostly defined in terms of itself; see Flatt et al. (2012).

Macros
let

- Exercise 1–2, 4. d), solutions p. 9. A **let**:

```
let ([var value] ...)
  body
  ...
```

is equivalent to a **λ**, called immediately:

```
((λ (var ...) body ...) value ...)
```

but having the [var init] specifications appear in one place – instead of separated by the function body, which may be long – makes the code much more readable.

- Racket also provides a similar convenience form to help write macros – with-syntax [1] [2] binds *syntax patterns* in a let-like fashion.

(This latter point is mainly of interest if you want to explore macro writing; useful to keep in mind that "there was a **let** for this" for when you eventually need it.)

- Difference between **let**, **let***, **letrec**:
  - **let**: parallel binding
    - Just syntactically; not concurrent.
  - **let***: sequential binding
  - **letrec**: (mutually) recursive binding

*While **let** makes its bindings available only in the bodys, and **let*** makes its bindings available to any later binding expr, **letrec** makes its bindings available to all other exprs—even earlier ones. In other words, **letrec** bindings are recursive.*

–TRG 4.6.3: Recursive Binding: letrec

- This is why named let uses **letrec**; the loop must have access to its own name to be able to call itself for the next iteration.

- **letrec** can be used to locally define (mutually) recursive functions.
  - If you Common Lisp, this use of **letrec** corresponds to labels.

Macros

Promises: **delay**, **force**

- ***Promise***: *delayed evaluation*.

```
require syntax/parse/define

define-syntax-parser delay
 (_ expr …)
  syntax
   λ () (expr …)

define force(promise)
 promise()
```

- Usage:

```
define my-promise
 delay
  displayln 'hello

;; …possibly much later…
force my-promise
```

⚠ Barebones implementation! Racket provides much better promises, with automatic memoization of results.

- delay makes a ***thunk***, i.e. a 0-argument function, representing a *promise* to compute something later.
  - Because a function runs only when called, this delays evaluation of the given code; hence the name delay.
- In an eager language (e.g. Python and most Lisps including Racket), delay needs to be a macro.
  - If delay was a function, the given code would be immediately evaluated (because it is then a function argument) before delay had a chance to make a thunk out of it.
  - Why define delay at all? **For humans**. We want to convey the intended meaning, not the details of implementation – and delay, force do that much better than a raw λ and a call into it.
    - *Package the design pattern as an abstraction.*
- The counterpart of delay, called force – meaning “compute it now” – can be a function, so there is no point in making it a macro.

Macros

and, or

- Essentially, sugar on top of nested **if**s:

```
require syntax/parse/define

define-syntax-parser or
  (_ a b more ...+)   ; ...+: one or more
   #'(or (or a b) more ...)
  (_ a b)
   #'(if a a b)  ; if test true-expr false-expr
  (_ a)
   #'(a)
  (_)
   #'(#f)

define-syntax-parser and
  (_ a b more ...+)
   #'(and (and a b) more ...)
  (_ a b)
   #'(if a b #f)
  (_ a)
   #'(a)
  (_)
   #'(#t)
```

- In Racket, if is a special case of **cond**.
  - Usually, it is more idiomatic to use **cond**.
  - Different Lisps, different preferences.
- These evaluate left-to-right, and return truthy or #f.
- Racket automatically applies the rule for a reducible case recursively, until the process “bottoms out” into the base cases.
- They need to be macros to avoid evaluating *b*, if *a* already fully determines the result.
- Testing:

```
(and #t #t #f)
(or #f 'hello 42)
```

⚠ Racket already provides **and** and **or**. The ones given here are meant only to illustrate how **and** and **or** desugar into a sequence of nested **if**s.

Macros

and, **or**

- To avoid an extra evaluation:

```
require syntax/parse/define

define-syntax-parser or
  (_ a b more ...+)
   #'(or (or a b) more ...)
  (_ a b)
   #'(let ([value a])
       (if value value b))
  (_ a)
   #'(a)
  (_)
   #'(#f)
```

The only change: evaluate “a” once, and bind a name to the result.

- This is more efficient, because the body of the two-argument base case of **or** may need *a* twice, and *a* may be a compound expression.

  - Keep in mind that this is a macro, so in effect, we are splicing source code.

    (Or, more precisely, editing the AST. Being based on *syntactic* macros, this is much less error-prone than if we tried to do something similar in the C preprocessor – in those very few cases where that is at all possible.)

- The body of **and** mentions each argument only once, so it is efficient as-is.

Macros

Infix math

- One final example. Lisps have no infix math by default, and sweet-exp sidesteps the issue by providing a hook for a custom macro called *nfx*.

- Opinions vary on what features are desirable:
  - With operator precedence, or simple left-to-right?
  - Support only basic arithmetic, or try to support anything that could be considered an operator?

- There are many ways to do this, including this quick hack (left-to-right) and the very advanced User-programmable infix operators in Racket.

- A simple one with precedence, from the Learning Racket blog series by Artyom (with minor edits):

```
require syntax/parse/define

define-syntax-parser nfx #:datum-literals (+ - * / ^)
    (_ left ... + right ...)  #'(+ (nfx left ...) (nfx right ...))
    (_ left ... - right ...)  #'(- (nfx left ...) (nfx right ...))
    (_ left ... * right ...)  #'(* (nfx left ...) (nfx right ...))
    (_ left ... / right ...)  #'(/ (nfx left ...) (nfx right ...))
    (_ left     ^ right ...)  #'(expt left      (nfx right ...))
    (_ x)               #'x

nfx 1 + 3 - (+ 1 2 3) * 4 / 5 + 3 * 7 ^ 2 ^ 2 * 5 - 3 / 5 / 8 - 9 / 7    ;; 36012 47/56

{1 + 3 - (+ 1 2 3) * 4 / 5 + 3 * 7 ^ 2 ^ 2 * 5 - 3 / 5 / 8 - 9 / 7}  ;; sweet-exp hook: mixed ops, {} → nfx
```

Exponentiation is right-associative.

Macros

Breaking hygiene

Maybe advanced...ish?

See [1] and [2].

- A standard silly example: ***the anaphoric if***.
  (anaphora: *referring to a preceding expression*.)

  We need a literal “it” to be available in the user code that calls the macro; but hygienic macros are specifically designed to prevent name leakage. So we need a special technique to expose “it”, while still reaping the benefits of hygienic macros:

```
require syntax/parse/define

define-syntax it(stx)
  raise-syntax-error 'it "only meaningful inside an aif"

define-syntax-parser aif
  (_ test if-true if-false)
   with-syntax ([it (datum->syntax #'test 'it)])
    syntax
     let ([it test])
      if test
       if-true
       if-false

aif (* 3 7) (* 2 it) 'nope
```

- **define-syntax** gives “it” a transformer binding – i.e. makes a syntax transformer – that by default (outside any “aif”) just raises a syntax error.
  - *This is how to make reserved words that may only appear inside certain macros.*
  - This catches invalid uses early.
- **with-syntax** binds a syntax pattern in a **let**-like fashion, allowing us – *inside the RHS of our macro definition* – to refer to the bound value by the *syntax pattern* “it”.
- For the value, we say that in the code, its textual form is “it” (the symbol 'it), and give it a *non-default lexical context* with datum->syntax.
- Copying the lexical context from the syntax object captured by the pattern *test*, we “leak” (*break hygiene on*) this ”it”, intentionally exposing it to user code *at the macro call site*.
- Finally, the **let** actually binds this “it” to a value.
  - Here we're inside a syntax template, so we can use it like any captured pattern.

Macros

Maybe advanced...ish?

Literal reserved words in Racket

- How to make literal reserved words, like Python's **in** in expressions like **for** x **in** range(…):

```
require syntax/parse/define

define-syntax bar(stx)
  raise-syntax-error 'bar "only meaningful inside a went-to-bar"

define-syntax-parser went-to-bar #:literals (bar)
  (_ a bar b)
    syntax
      displayln
        format
          "~a went to a ~a"
          a
          b

(went-to-bar 'tom bar 'pub)
(went-to-bar 'jerry bar 'tavern)

;; syntax error: expected the identifier 'bar'
(went-to-bar 'jerry tavern 'tavern)
```

The macro was originally called foo, hence a literal bar; but the Racket Style Guide recommends using semantically meaningful names in code examples.

- We tell **define-syntax-parser** that if the word “bar” appears in the syntax pattern being matched, it is to be taken literally.
  - Without the #:literals, the syntax parser would match the “bar” to *anything*, and let us refer to it as *bar* – great for default behavior, but not what we want here.
- This requires a binding. Almost always you want that, but when not, use #:datum-literals instead of #:literals.

Macros in Python

- Macropy for Python 3.4+, with examples and how to write your own macros.
  - For documentation on the Python AST, see the standard library docs and Green Tree Snakes.
  - ⚠ Not as easy to write your own macros as in Lisps; Python's syntax is not homoiconic, and is much more complex than Lisp's.
  - ⚠ Writing macros is a subset of programming language design – “easy” is relative.

- Example – memoizing thunk (like Racket's delay, force):

```
from macropy.quick_lambda import macros, lazy

# count how many times expensive_func runs
count = [0]
def expensive_func():
    count[0] += 1

thunk = lazy[expensive_func()]

print(count[0] # 0)

thunk()
print(count[0]) # 1
thunk()
print(count[0]) # 1
```

Import the magic name **macros**, and then any macros you want to use.

General usage:

```
from my_macro_module import macros, …

val = my_expr_macro[…]

with my_block_macro:
    …

@my_decorator_macro
class X():
    …
```

⚠ MacroPy macros run ***at import time*** (using PEP302 import hooks), so cannot be used in a file that is run directly. See the minimal viable setup in MacroPy documentation.

let *over* ***λ***

A Lisp idiom

Language-specific design pattern; typically so small that there is no point in extracting it as a macro.

- A puzzle – what does this code do?

```
define f
 let ([x 0])
  λ ()
   set! x {x + 1}
   x
```

Here "the **let** hangs over the λ"; hence the name.

In Racket, see also splicing-let.

- Defines a function f, as we saw further above, discussing **define** and λ.
  - *Correct, but not the main point here.*

- The trick is in the **let** sandwiched between the **define** and the λ – it creates a local name x… outside the function body.
  - Lexically scoped, as usual: this x is not visible outside the **let** block.
  - The **let** is set up first, before the λ is bound to the name f. The body of f() starts just below the λ. Thus, the **let** runs only once (**not** at each call of f).
    - Hence, in this example, f counts how many times it has been called.

- ***let over λ: local data that persists across calls to the function!***
  - For now the interaction with it is very limited, but that's easily fixed…

let over λ

Introducing the **dispatcher**

- Read-and-write local data, using a ***dispatcher***:

```
define f
 let ([x 0])
  define set!-x(value) (set! x value)
  define get-x() x
  define call()
   set!-x {x + 1}
   x
  λ (msg)
   cond
    (eq? msg 'set!-x)
     set!-x
    (eq? msg 'get-x)
     get-x
    else  ; default
     call

define f-set!-x (f 'set!-x)
define f-get-x  (f 'get-x)
define f-call   (f 'call)
```

Dispatcher: take a message, return the corresponding function. This gets bound to the name f.

Just a symbol; no relation to the function with the same name.

- Testing it:

```
f-call()        ; 1
f-call()        ; 2
f-call()        ; 3
f-get-x()       ; 3
f-set!-x(42)
f-get-x()       ; 42
f-call()        ; 43
f-call()        ; 44
f-call()        ; 45
```

Could also use **case** instead of **cond** to implement the dispatcher; refer to the Racket docs on case.

This dispatch technique is also known as *message passing* (no relation to MPI).

- *Methods*? Starting to look a bit like OOP?

*λ over **let** over λ*

Constructor; instance variables

- How about some OOP?

```
define Myobj()
  let ([x 0])
    define set!-x(value) (set! x value)
    define get-x() x
    define call()
      set!-x {x + 1}
      x
    λ (msg)
      cond
        (eq? msg 'set!-x)
          set!-x
        (eq? msg 'get-x)
          get-x
        else  ; default
          call
```

Constructor, now a function.
Its args would go here.

The constructor returns the **dispatcher**, which is *a functional representation of the object instance*.

(Keep in mind that when returning a function, what actually gets returned is a *lexical closure*.)

- Now we can construct Myobj instances, (usage omitted; works the same as for f above):

```
define myobj1 (Myobj)
define myobj1-set!-x  (myobj1 'set!-x)
define myobj1-get-x  (myobj1 'get-x)
define myobj1-call    (myobj1 'call)
```

Now the **let** re-runs each time the constructor is called; hence, each created object instance gets its own copy of x.

⚠ Observe that this is built out of the “fundamental bricks” of Lisp; there is no reference to any existing object system.

⚠ In the real world, when writing an OO program in Racket, see **struct** and **class**; Racket already provides OOP!

λ over *let* over λ in *let* over λ ?

Class variables; OOP in FP

- Refining our OOP a bit:

In Python terms, a type object constructor.

```
define Myobj-typeinit(s0)
 let ([s s0])                ; class (“static”) variables
  define set!-s(value) (set! s value)
  define get-s() s
  define init(a0)          ; instance constructor
   let ([a a0])            ; instance variables
    define set!-a(value) (set! a value)
    define get-a() a
    define call()
     set!-a {a + 1}
     a
    λ (msg)     ; instance dispatcher
     cond
      (eq? msg 'set!-s) set!-s
      (eq? msg 'get-s) get-s
      (eq? msg 'set!-a) set!-a
      (eq? msg 'get-a) get-a
      else call
  λ (msg)        ; class dispatcher
     cond
      (eq? msg 'set!-s) set!-s
      (eq? msg 'get-s) get-s
      else init
```

Lexical scoping.

- Testing it:

```
define Myobj Myobj-typeinit(42)
define myobj1 (Myobj 'init)(17)
define myobj2 (Myobj 'init)(23)

(myobj1 'get-a)()  ; 17
(myobj2 'get-a)()  ; 23
(myobj2 'call)()    ; 24
(myobj1 'get-s)()  ; 42
(myobj2 'get-s)()  ; 42
(Myobj 'get-s)()    ; 42
(myobj1 'set!-s)(9001)
(myobj2 'get-s)()  ; 9001; class variable.
```

- Full example on GitHub.

⚠ Note how class variables live *in a surrounding lexical scope*, as seen from the viewpoint of an object instance.

We'll stop this here, but for more, see
Doug Hoyte (2008): Let over Lambda: 50 Years of Lisp,
a koan about objects and closures, and
OOP in FP with inheritance (delegation is inheritance!).

Continuations

Not to be confused with the mathematical concept.

- Partial motivation:
  - Macros alone would be useless; a means of abstraction needs primitives.
  - We have seen, roughly, how to build – out of the basic bricks of Lisp – simple versions of things such as loops, promises, logical operators, and objects.
  - How about things that require more general control flow, e.g. **return** or generators?
  - Enter the…

- ***Continuation***:
  - The remaining steps in a computation. –Matthew Might: Continuations by example
  - The control state of the program. –TRG 10.3: Continuations; Wikipedia

- See also Matthew Butterick: Beautiful Racket: Continuations.
- Or see this story about a sandwich; or this one about continuations and coffee.

- In plain English, a continuation is a bookmark to a point in the program's execution, making it possible to jump back to that point later.
- When jumping back to such a “bookmark”, only the control flow resets. Any mutations to the state of the program's data will remain – this is what makes it useful.
- Modern variant: *delimited continuations.*

- Possible applications: **return**, **while**, backtracking search, generators, threads, coroutines, exceptions.

call/cc

- **call/cc** (a.k.a. *call-with-current-continuation*) is a primitive that takes as its argument a function, and gives to that function the current continuation as a first-class object. It is a low-level “brick” for creating control flow constructs.

- Why would anyone want to ***call*** anything ***with*** the cc? *What does that even mean?*

  - It's a lispy API that leans on the composability of the syntax and “lambdas as code blocks”:

    ```
    call/cc
      λ (cc)
        …  ; in this block cc is the current continuation
    ```

    From the viewpoint of understanding this code, the λ is really just a code block that will run immediately. The fact that it also happens to be a function is immaterial.

  - No need for parameters other than cc – any free variables in the λ block are closed over by lexical scoping!

  - Python **lambda**s do not lend themselves to this programming style (perhaps on purpose).

  - The call-with-something is an idiom to insert a handler to run before a particular code block is entered. The handler passes the “something” as an argument into that code block.

    - Lexical scoping then ensures that the name originally bound to the “something” is only accessible inside that code block.

    - But it can be set!'d into something visible from outside, or returned when the block ends.

call/cc

- The continuation bookmarks the point where the call/cc ends.

- When the continuation – which can be called as if it was a function – is called, program execution will jump to the bookmarked point.

- Hence, calling (cc) while still inside the block will exit the block. It behaves like a **break**.
  - If this is all you need to do, there is also a more limited call/ec (*call-with-escape-continuation*), which gives better performance.
  - The difference is that the ec must be called (if it is to be called at all) *within the dynamic extent* of the call/ec'd block, whereas a cc may be called *any time later*.
  - An ec is slightly more general than a **break**. With a properly placed call/ec, you can break out of multiple nested loops *with one call*.
    - Actually, **continue**, **break** and **return** are ecs, recorded at different points.

- When called later, a continuation allows jumping back to an already visited point in the code.

- The first time, on normal exit, call/cc returns whatever the λ block returns.
- When the continuation is invoked, any arguments given to the continuation, as in (cc arg1 arg2 … argn), become the return value(s) of the call/cc.

- There is also a **let/cc**, which can sometimes be more readable:

```
let/cc cc
  …  ; in this block cc is the current continuation
```

  This bookmarks the point where the **let** block ends.

Implementing **return**

- Lisps don't have a **return** statement, but we can add one with call/cc (full example):

(Useful for returning early, i.e. before falling through the end.)

```
require syntax/parse/define

define-syntax-parser define/return
  (this-stx (f args ...) body ...)
    with-syntax ([return datum->syntax(#'this-stx 'return)])
      syntax
        define (f args ...)
          let/ec return
            body
            ...
```

We have chosen the name **define/return** for this syntax to express the idea "define a function that uses **return**".

This code is spliced to the macro call site. We have inserted a **let/ec**; otherwise it's just a regular **define**.

```
define-syntax-parser λ/return
  (this-stx args body ...)
    with-syntax ([return datum->syntax(#'this-stx 'return)])
      syntax
        λ args
          let/ec return
            body
            ...
```

The same for a λ that uses **return**.

Usage:

```
define/return f()
  displayln "hi"
  return 42
  displayln "not reached"
```

- A **while** loop can be defined similarly, making appropriate bookmarks for **continue** and **break**.
- Matthew Might: You don't understand exceptions, but you should
  - *To understand exceptions is to implement exceptions.*
  - **return**, **while**, exception handling with call/cc. These interact! (Advanced but enlightening.)

Demystifying the details of

Implementing **return**

- Lisps don't have a **return** statement, but we can add one with call/cc (full example):
  (Useful for returning early, i.e. before falling through the end.)

```
require syntax/parse/define

define-syntax-parser define/return
  (this-stx (f args ...) body ...)
    with-syntax ([return datum->syntax(#'this-stx 'return)])
      syntax
        define (f args ...)
          let/ec return
            body
            ...
```

Break hygiene, i.e. leak the name "return" to user code at the macro call site.

The magic: bind the name "return" to the *escape continuation*; i.e. bookmark the point where this block ends.

User code goes here.

```
define-syntax-parser λ/return
  (this-stx args body ...)
    with-syntax ([return datum->syntax(#'this-stx 'return)])
      syntax
        λ args
          let/ec return
            body
            ...
```

Usage:

```
define/return f()
  displayln "hi"
  return 42
  displayln "not reached"
```

This invokes the continuation. The result of the **let/ec** block, and hence also of the function, becomes whatever we give here as arguments to **return**.

- A **while** loop can be defined similarly, making appropriate bookmarks for **continue** and **break**.
- Matthew Might: You don't understand exceptions, but you should
  - *To understand exceptions is to implement exceptions.*
  - **return**, **while**, exception handling with call/cc. These interact! (Advanced but enlightening.)

Implementing generators

- Maybe more interesting from a Python viewpoint: *how does a generator work*? (Full example.)

```
require syntax/parse/define

define-syntax-parser make-generator
  (_)
   raise-syntax-error 'make-generator "missing body"
  (this-stx body ...)
   with-syntax ([yield (datum->syntax #'this-stx 'yield)])
    syntax
     let ([k 'none])
      λ/return ()
       let ([yield (λ args
                    (let/cc cc
                     (set! k cc)
                     (apply return args)))])
        cond
         (not (eq? k 'none))  ; resume?
          k()
         body
         ...
```

let over λ!

Kontinuations are konventionally kalled *k*.

Again, this is what gets spliced to the macro call site.

⚠ Like Python, Racket already provides generators.

Usage:

```
define g
 make-generator
  let loop ([x 42])
   yield x
   loop {x + 1}
g()  ; 42
g()  ; 43
g()  ; 44
```

- See also Matthew Might: Continuations by example
  - Backtracking search, generators, threads, coroutines.

Demystifying the details of

Implementing generators

- Maybe more interesting from a Python viewpoint: *how does a generator work*? (Full example.)

```
require syntax/parse/define

define-syntax-parser make-generator
  (_)
   raise-syntax-error 'make-generator "missing body"
  (this-stx body ...)
   with-syntax ([yield (datum->syntax #'this-stx 'yield)])
    syntax
     let ([k 'none])
      λ/return ()
       let ([yield (λ args
                    (let/cc cc
                     (set! k cc)
                     (apply return args)))])
        cond
         (not (eq? k 'none))  ; resume?
          k()
        body
        ...
```

⚠ Like Python, Racket already provides generators.

The usual fandango to make **yield** visible.

Current bookmark.

Since **yield** kinda-is-a **return**, we base this on our **return** implementation.

Magic, vol. 1: bookmark the place just after the current **yield**.

Magic, vol. 2: when we are called, check whether we have an active bookmark, and resume if so.

Usage:

```
define g
 make-generator
  let loop ([x 42])
   yield x
   loop {x + 1}
g()  ; 42
g()  ; 43
g()  ; 44
```

- See also Matthew Might: Continuations by example
  - Backtracking search, generators, threads, coroutines.

What else could we do with **call/cc**?

- How about some declarative programming – ***nondeterministic evaluation***? [1] [2] [3] (Full example.)

```
require "choice.rkt"

choice 1 2 3  ; 1
next()  ; 2
next()  ; 3
next()  ; #f

all {choice(3 10 6) + choice(100 200)}  ; '(103 203 110 210 106 206)

all-query(even? {choice(4 5) + choice(11 14)})  ; '(18 16)

let ([a (choice 1 2 3 4 5 6 7)]
     [b (choice 1 2 3 4 5 6 7)]
     [c (choice 1 2 3 4 5 6 7)])
  assert (= (* c c) (+ (* a a) (* b b)))
  displayln (list a b c)  ; (3 4 5)
  assert (< b a)
  displayln (list a b c)  ; (4 3 5)
```

- Based on David Liu (2014): Continuations and Backtracking in Racket.
- The Pythagorean triples example is from Matthew Might: Continuations by example.
- SICP approaches this problem from the interpreter side, showing how to build the choice operator *amb* into the language core, while demonstrating continuation-passing style.
- If you seriously need this sort of thing in Racket, consider Racklog: Prolog-Style Logic Programming.

Continuations in Python

- continuation (any version of Python) implements *continuation-passing style* with simple syntax:

- Example: tail-recursive factorial with the *continuation* library:

```
from continuation import with_CC, with_continuation

@with_continuation
def k_factorial(k):
    def inner(n, acc):
        return acc if n < 2 else (k << k_factorial)(n - 1, n * acc)
    return inner
factorial = lambda n: (with_CC >> k_factorial)(n, 1)

print(factorial(7))
```

Explicit **return** as usual in Python.

This library doesn't add **call/cc**. It's likely that could be done with Python's powerful introspection features (by manipulating the call stack directly), but the author has chosen another, perhaps somewhat more robust approach.

- *with_CC* >> *func* sets up a continuation object. Execution starts from *func*, which must be decorated by *@with_continuation*.
  - *func* will receive one argument, the continuation object.
  - Feed other arguments using the above pattern.

- *k* << *func* sends the function *func* to *k* to use as the continuation – i.e. as what happens next; ***the remaining steps in the computation***. Call that (like above) to jump to the continuation.

- This *trampolines* [1] [2] [3] automatically, so it won't overflow the call stack.

- For *tail call optimization* (TCO), there is another library from the same author…

FP loop with TCO

More on the *tco* library on the next slide.

```
from tco import C, with_continuations

def looped_hello(n):
    @with_continuations()  # TCO
    def loop(i, self=None):
        print("hello from loop, i = {}".format(i))
        return self(i - 1) if i > 1 else None
    return loop(n)  # start the loop

looped_hello(5)
```

```
define looped-hello(n)
  let loop ([i n])  ; named let; starts implicitly
    displayln
      format
        "hello from loop, i = ~a"
        i
    cond
      {i > 1}
        loop {i – 1}

(looped-hello 5)
```

More continuations in Python

- ***Tail call optimization*** is provided by tco (any version of Python).

- Example: tail-recursive factorial with the *tco* library:

```
from tco import C, with_continuations  # "C" is the low-level API; see documentation.

def factorial(n):
    @with_continuations()
    def inner(n, acc, self=None):
        return self(n - 1, n * acc) if n > 1 else acc
    return inner(n, 1)  # start the loop

print(factorial(7))
```

To apply TCO, the call must be in *tail position*.

Tail position: the last thing the function does before **return**ing (in a particular code path).

⚠ Not necessarily last in the source code of that function; **return**ing earlier is also allowed.

- The function *inner* is a functional loop. **With *tco*, FP loops work properly in Python**!
  - *n* counts down how many steps are remaining, *acc* is the accumulator.
  - ***This example uses no mutation***! Each iteration of the functional loop gets a fresh *n* and *acc*.

- The continuation *self* is used for tail-call-optimized recursive calls.

  - Its default value can be anything; it is not used. (But **None** is an obvious placeholder.)
  - The *@with_continuations* decorator sets up the continuation when the function is called.

  - Different from the usual use of *self* in Python!
    - It's not the first argument; instead, it's passed by name. Indeed, it's not Python's *self.*
      It's just a parameter that happens to be called "*self*", having somewhat similar semantics.

More continuations in Python

- In effect, the tco library, in a manner of speaking, also adds **call/ec** to Python.
- How to use: beside *self*, more continuations can be given, e.g. to be called in success and failure cases; or to be called **to escape from multiple levels of nested loops**.
- Example: tail-recursive factorial, using an escape continuation to finish.
  - There is only one level of looping in this example, but this illustrates how to use an ec.

```python
from tco import C, with_continuations

def factorial(n):
    @with_continuations()  # let's make a continuation...
    def identity(x, self=None):
        return x
    @with_continuations(ec=identity)            # bind any extra continuations here…
    def inner(n, acc, self=None, ec=None):  # …and just declare them here.
        return self(n - 1, n * acc) if n > 1 else ec(acc)
    return inner(n, 1)
```

- We set up a function, *identity()*, that we can use as an ec in continuation-passing style.
  - The mind-bending part: the **return** in *identity() returns from the **original** top-level call!*
  - This is basically because our function *identity()* is a continuation.
- See:
  - GitHub page of the tco library
  - Thomas Baruchel: Explaining functional aspects in Python, an explanation from the author.

Breaking out of a code block

- OTOH, **continue**, **break** and **return** *already are second-class escape continuations*.

- **Taking a page from Lisps**: *how to set up a custom "escape bookmark" in pure Python?*
  - We can do this with a not-really-a-function, to run immediately – like a Lisp λ as a code block.
  - Abuse **def**, **return**, and decorators!

```
def immediate(thunk):  # decorator to replace a thunk definition by its result.
    return thunk()

def main():
    @immediate    # the grammar insists on a newline ("@immediate def" would sound nicer).
    def result():
        return "hello"  # return to escape (possibly early) from the block and return a value.
    # now result is the string "hello".
```

We could also name this decorator *call_ec*, which it *almost* is. (Almost, because **return** is not a first-class value.)

- Compare the usual use of decorators (note caveats!):

```
def deco(f):
    def decorated(*args, **kws):
        print("deco says hi!")
        return f(*args, **kws)
    return decorated

@deco
def myfunc():
    print("hello from myfunc")
```

Closed over the free var *f*.

The factory returns a decorator Python then implicitly calls to actually decorate the user function.

Same as
def myfunc(): …
myfunc = deco(myfunc)

⚠ Need arguments? Factory-wrap it:

```
def make_deco(a, b):
    def deco(f):
        def decorated(*args, **kws):
            print("deco says {}, {}".format(a, b))
            return f(*args, **kws)
        return decorated
    return deco

@make_deco(17, 23)
def myfunc():
    print("hello from myfunc")
```

Pattern matching

Not to be confused with string matching or pattern recognition.

- **What**: A conditional based on shape and value of input data; seen in FP languages.
  - Destructuring bind, with a vengeance – switch on structure of input, while naming its parts.
  - Function overloading *by argument value*, not only by argument type.

  - Contrast *string matching*: pattern matching operates on structured data, such as expressions.
  - Contrast *pattern recognition*: a pattern match must be exact.

  - Commonly used in Haskell; comboes well with *ADTs* (*algebraic data types*).
  - Something of an introduced species in untyped languages. Racket has an extensible **match**.
    - Also **syntax-parse** uses pattern matching, but the syntax is slightly different.

- **Why**: *Code is for humans* – pattern matching improves readability. A simple destructuring:

Traditional lispy solution.

```
define foldl(f x lst)
  cond
    empty?(lst) x
    else (foldl f (f (car lst) x) (cdr lst))

define foldr(f x lst)
  cond
    empty?(lst) x
    else (f (car lst) (foldr f x (cdr lst)))
```

(Could name the parts, but needs more code.)

Docs: guide, ref

With pattern matching.

```
define foldl(f x lst)
  match lst
    '() x
    (cons a l) (foldl f (f a x) l)

define foldr(f x lst)
  match lst
    '() x
    (cons a l) (f a (foldr f x l))
```

Name the parts right away.

"If *lst* is a *cons* of *a* and *l*, then..."

Pattern matching

- Function overloading *by value* (full example):

Traditional lispy solution.

```
define factorial(n)
 let loop ([m n]
           [acc 1])
   cond
     {m = 1} acc
     else (loop {m - 1} {m * acc})

(factorial 4)
```

With pattern matching.

```
define factorial(n)
 let loop ([m n]
           [acc 1])
   match m
     1 acc
     v (loop {v - 1} {v * acc})

(factorial 4)
```

Match anything;
bind identifier.

- In both solutions, we have used the same linear process as in the Python continuation example:
  - Functional loop.
  - *m* keeps track of remaining steps, *acc* is the accumulator.

- We are essentially overloading *loop* to do different things based on the **value** of *m*.
  - On the other hand, we're essentially using PM as a **cond** with fancy syntax.

- To demonstrate the full power of this approach, we would need a typed language and something like *algebraic data types*.
  - Can be done in typed/racket.

- If none of the given patterns match the given input, it is an error.
  - In static typing, the type system may catch missing cases, issuing a compile-time warning [1] [2].
  - In dynamic typing, runtime error.

Pattern matching in Python

- **Wait, doesn't Python already do that?** Almost, but not quite:

  - Python supports a limited form of destructuring bind with the tuple unpacking operator, *
    - Can pull from nested tuples, like in the *zip-as-an-unfold* example in lecture 10, slide 14.
    - Tuples, lists and namedtuples only. No generators (maybe reasonable), no custom objects.

  - *But Python doesn't have a concise way to match different input data structure shapes.*
    - Important for choosing the correct destructuring for variably shaped input.

- **Libraries**:
  - MacroPy has PM; example:

    Case classes [1] [2] [3].

    Roughly speaking, here used to approximate an ADT.

    Look familiar?

```
from macropy.case_classes import macros, case
from macropy.experimental.pattern import macros, switch

@case
class Nil():
    pass

@case
class Cons(x, xs):
    pass

def reduce(op, my_list):
    with switch(my_list):
        if Cons(x, Nil()):
            return x
        elif Cons(x, xs):
            return op(x, reduce(op, xs))

print(reduce(lambda a, b: a + b, Cons(1, Cons(2, Cons(4, Nil())))))  # 7
print(reduce(lambda a, b: a * b, Cons(1, Cons(3, Cons(5, Nil())))))  # 15
print(reduce(Nil(), lambda a, b: a * b))  # None
```

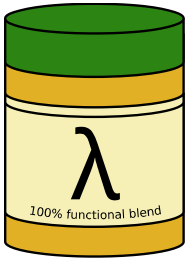
Pattern matching in Python

- For PM only, there's also PyPatt: Python Pattern Matching.
  - But untested with Python 3; might not currently work.
  - See its PyPI page for links to alternatives and further reading.

- Example from the README (converted to Python 3 syntax):

```
import pypatt

@pypatt.transform
def test_demo():
    values = [[1, 2, 3], ('a', 'b', 'c'), 'hello world',
        False, [4, 5, 6], (1, ['a', True, (0,)], 3)]
    for value in values:
        with match(value):
            with 'hello world':
                print('Match strings!')
            with False:
                print('Match booleans!')
            with [1, 2, 3]:
                print('Match lists!')
            with ('a', 'b', 'c'):
                print('Match tuples!')
            with [4, 5, quote(temp)]:
                print('Bind variables! temp =', temp)
            with (1, ['a', True, quote(result)], 3):
                print('Nest expressions! result =', result)
    print('Wow, pretty great!')
```

Automatic currying



- Recall *currying* (lecture 10, slide 17). *What if the language did that automatically?*

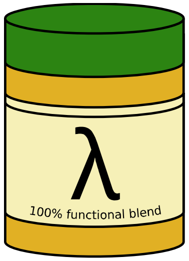
```
#lang sweet-exp spicy

define reverse
  foldl cons empty

define append(a b)
  foldr cons b a

define sum
  foldl + 0

define product
  foldl * 1

define map(f)
  foldr (compose cons f) empty

define sum-matrix
  compose sum (map sum)
```

```
module+ main
  define a '(1 2)
  define b '(3 4)
  define c '(5 6 7)
  define M '((1 2)
             (3 4))
  define f(x) {x * x}
  append a b
  reverse c
  sum a
  product b
  product (append a b)
  map f c
  sum-matrix M
```

Full example on GitHub.
Same example without auto-currying.

- Examples taken from Hughes (1984): Why Functional Programming Matters and racketified.

Automatic currying



- Recall *currying* (lecture 10, slide 17). *What if the language did that automatically?*

Point-free style (PFS): omit declaring the arguments when possible.

```
#lang sweet-exp spicy

define reverse
  foldl cons empty

define append(a b)
  foldr cons b a

define sum
  foldl + 0

define product
  foldl * 1

define map(f)
  foldr (compose cons f) empty

define sum-matrix
  compose sum (map sum)
```

Usage is (foldl f x lst) so this becomes a function of one argument.

Note the arity mismatch; *f* is expected to take 1, return 1, whereas *cons* takes 2, returns 1.

The extra arg to *f* is "passed through" on the right.

```
module+ main
  define a '(1 2)
  define b '(3 4)
  define c '(5 6 7)
  define M '((1 2)
             (3 4))
  define f(x) {x * x}
  append a b
  reverse c
  sum a
  product b
  product (append a b)
  map f c
  sum-matrix M
```

Racket's conditional main construct.

Full example on GitHub.
Same example without auto-currying.

- Examples taken from Hughes (1984): Why Functional Programming Matters and racketified.

Meta

The finish line

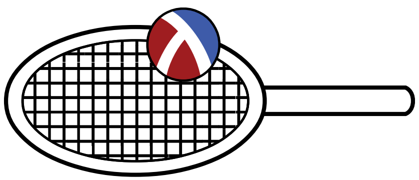
- Racket code examples in this lecture (plus a bunch more) are available on GitHub, in a subfolder of examples, named beyond Python.

  - We didn't talk (much) about:
    - Infinite streams [1] [2] (custom SICP style; in production, use Racket's).
    - Simple infix syntax processor (manually implemented, left-to-right).
    - Sieve of Eratosthenes in Racket (several implementations).
    - Quicksort: functional; imperative; Python for comparison.
    - typed/racket: a simple gcd, parametric types, refined types.
    - Extra examples of pattern matching [1] [2].

  - spicy (automatic currying for Racket) is provided as a separate project.

- Accompanied by this set of slides, the examples explain the basics, including some beginner and intermediate macrology.

  - Intended as a starting point for self-study for the adventurous.



In conclusion

The future of programming – the next 30 years?

- *Prediction is very difficult, especially about the future.*
  –Niels Bohr? [disputed]

- Functional programming is rising. But which language will win? (Year of first release in parens.)
  - Haskell (1990): highly advanced statically typed language, pure FP.
  - Lisp? May be cursed; “worse is better“.
    - Racket (1994): production-quality; excels at extensibility; highly modular ecosystem.
    - Clojure (2007): currently well positioned; runs on the Java VM and integrates with Java.
  - A black swan?

- OOP may be on the way out, very slowly. Widely adopted, but mostly incompatible with pure FP. OOP knowledge still needed to maintain legacy code.

- Languages will continue to rise and fall; no one language captures all of programming.
- Fortran will likely continue surviving as a special-purpose language for low-level numerics.
- C will likely remain the language of choice for implementing operating systems.
- Many Lisps will likely continue surviving, for teaching and for thinking about languages.
- Python? 1.0 in 1991; too early to tell. Has found its niche as an easy, powerful, general-purpose high-level language, but a new unexpected contender replacing it cannot be ruled out.

- Parallelization paradigms will continue to evolve. Massively parallel supercomputing will likely remain relevant in scientific computing.

- Quantum computing may revolutionize some fields of application, but the programming model is so fundamentally different (applied quantum mechanics!) that algorithms must be completely reinvented to be able to use QC – a much harder task than the transition to the parallel model.

In conclusion

Meanwhile – which language?

- If you want…
  - Ease of use, power plant included ⇒

    Someone has even written this…

    https://www.python.org/
  - A pure functional language with a highly advanced static type system ⇒

    Secret alien technology, vol. 2.

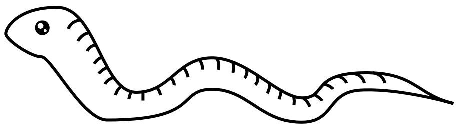
    https://www.haskell.org/
  - The pinnacle of the first 60 years of Lisp; to discover EP and LOP ⇒

    Programming language construction kit.

    (More useful than it sounds?)

    https://racket-lang.org

In conclusion



Final words

```
import logging
from datetime import datetime

logger = logging.getLogger(__name__)

try:
    while True:
        with Funding(datetime.now()) as f:
            do_science(f)
except RuntimeError as e:
    logger.error(e)
    from sys import exit
    exit(-1)
```

For example:
- What went well?
- What could be done better?
- What you would like to hear in a second course?

- Feedback? Questions? ⊳ juha.jeronen@tut.fi
- **For course credit**, submit your final assignment by **31.5.2018** 23:59:59.$\overline{9}$

Appendix: Racket resources

- **General**:
  - Quick: An Introduction to Racket with Pictures
  - More: Systems Programming with Racket
  - The Racket Guide
  - How to Program Racket: a Style Guide (the equivalent of Python's PEP8, for Racket)
  - learning-racket (a useful series of beginner to intermediate level blog posts)
  - Learn Racket in Y Minutes
  - The Racket Reference – see especially the Language Model.
  - Matthew Butterick: Beautiful Racket (how to build programming languages in Racket)
  - Rosetta Code (code snippets for the same tasks in many languages, including Racket)
  - The Computer Language Benchmarks Game (includes entries written in Racket)

- **Indentation sensitive syntax**:
  - ⚠ *For pythonistas, makes a lot of sense, but the Racket community prefers parentheses.*
  - SRFI-110: Sweet-expressions (t-expressions) (syntax reference)
  - Sweet: an alternative to s-expressions (Racket implementation)

- **Extending the syntax – macros**:
  - Fear of Macros (excellent introduction to macros, recommended)
  - Macros in the Racket Guide
  - Syntax: Meta-Programming Helpers
  - Syntax Parse Examples
  - Macros in the API reference

- **General, on the Scheme family**:
  - The Adventures of a Pythonista in Schemeland
  - Abelson and Sussman: SICP, 2nd ed. – also discusses SW design, interpreters, compilers.

Racket data structures

for Python users

Python	Racket	
list	gvector	require data/gvector
set	mutable-set	require racket/set
frozenset	set	require racket/set
dict	make-hash	
[no immutable dict type]	hash	
namedtuple, class	struct (granular control on mutability)	
class	class [guide] [reference]	require racket/class

Additionally:

pair
mpair: mutable pair
list: immutable singly linked list
mlist: mutable; see related functions
require compatibility/mlist
vector: fixed-size vector
box: minimal mutable storage

For dynamically scoped variables in Racket (like *dynscope.py* for Python), see parameterize.

See Datatypes in the Racket reference.