Outline

1. Guessing Secrets
   - functions returning functions
   - oracles and trapdoor functions
2. anonymous functions
   - lambda forms
   - map(), reduce(), filter(), eval(), and apply()
   - estimating π with list comprehensions
3. List Comprehensions
   - algorithms and data structures
   - sequences, dictionaries, lists
4. Summary + Assignments

MCS 260 Lecture 15
Introduction to Computer Science
Jan Verschelde, 15 February 2016

guessing secrets

A little game: try to guess a number.

Typical *repeat until*:

```
generate secret
repeat
    ask for a guess
until guess equals secret
```

This game is typical for password verification.

lambda forms
list comprehensions

functions returning functions

Instead of storing the secret explicitly, we use *an oracle*.

For a given input, the oracle will return True if the input matches the secret and return False otherwise.

Our number guessing game with an oracle:

```
oracle = generate_secret()
repeat
    guess = input('give number : ')
until oracle(guess)
```

The function `oracle()` is a function computed by the function `generate_secret()`.

lambda forms
list comprehensions

oracles and trapdoor functions

password security

Guarding of passwords on Unix:

- the password is encrypted,
- only the encrypted password is saved on file.

Password verification consists in

1. calling the encryption algorithm on user input,
2. checking if the result of the encryption equals the encrypted password stored on file.

The encryption algorithm acts as an oracle.

The oracle is typically a *trapdoor* function:

1. efficient to compute output for any input,
2. very hard to compute the inverse of an output.

lambda forms
list comprehensions

lambda forms (or anonymous functions)

Lambda forms are functions without name, syntax:

```
lambda < arguments > : < expression >
```

Often we want to create functions rapidly,
or to give shorter, more meaningful names.
For example, to simulate the rolling of a die:

```
>>> import random
>>> die = lambda : random.randint(1,6)
>>> die()
2
```

The function `die()` has no arguments,
but just as with any other function, lambda forms can have default values, keyword and optional arguments.

functions as objects

Use default arguments to extend `die()`:

```
>>> import random
>>> die = lambda a=1,b=6: random.randint(a,b)
>>> die()
2
>>> die(0,100)
34
>>> die(a=-100)
-29
```

Functions are also objects:

```
>>> type(die)
<type 'function'>
>>> die
<function <lambda> at 0x40247294>
```

functions returning functions

Recall the guessing of a secret. The secret itself is less important than its function: we want an oracle to separate those who know the secret from those who don't.

Our application is to return a function

```
def make_oracle():
    "returns an oracle as a lambda form"
    numb = int(input('Give secret number : '))
    return lambda x: x == numb
```

In the main program:

```
ORACLE = make_oracle()
```

the guessing game with a lambda form

```
def make_oracle():
    "returns an oracle as a lambda form"
    numb = int(input('Give secret number : '))
    return lambda x: x == numb

ORACLE = make_oracle()
while True:
    GUESS = int(input('Guess the secret : '))
    if ORACLE(GUESS):
        break
    print 'wrong, try again'
print 'found the secret'
```

lambda forms
list comprehensions

1. Guessing Secrets
   - functions returning functions
   - oracles and trapdoor functions

2. anonymous functions
   - lambda forms
   - map(), reduce(), filter(), eval(), and apply()
   - estimating π with list comprehensions

3. List Comprehensions
   - algorithms and data structures
   - sequences, dictionaries, lists

4. Summary + Assignments

the map() function to map functions to lists

map() performs the same function on a sequence, syntax:

```
map ( < function > , < sequence > )
```

where the `function` is often anonymous.

Enumerate all letters of the alphabet:

```
>>> ord('a')
97
>>> chr(97)
'a'
>>> map(chr,range(97,97+26))
['a', 'b', 'c', .. , 'y', 'z']
```

observe the use of range

combining lists with map()

Adding corresponding elements of lists:

```
>>> a = range(0,3)
>>> b = range(3,6)
>>> map(lambda x,y: x+y, a,b)
[3, 5, 7]
```

To create tuples, use `list()` on `zip()`:

```
>>> list(zip(a,b))
[(0, 3), (1, 4), (2, 5)]
```

Let us add the elements in the tuples of `m`:

```
>>> map(lambda x: x[0]+x[1], m)
[3, 5, 7]
```

the reduce() function

The function `reduce()` returns one single value from a list, for example to compute the sum:

```
>>> r = range(0,10)
>>> reduce(lambda x,y: x+y , r)
45
```

The function given as argument to `reduce()` must

- take two elements on input,
- return one single element.

`reduce()` repeatedly replaces the first two elements of the list by the result of the function, applied to those first two elements,
until only one element in the list is left

the filter() function

Filters a sequence, subject to a criterion, syntax:

```
filter ( < criterion > , < sequence > )
```

where `criterion` is a function returning a boolean,
and the `sequence` is typically a list.
The list on return contains all elements of the input list for which the criterion is True.

Sieve methods to compute primes:

```
>>> s = range(2,100)
>>> s = filter(lambda x: x%2 != 0,s)
>>> s = filter(lambda x: x%3 != 0,s)
>>> s = filter(lambda x: x%5 != 0,s)
>>> s = filter(lambda x: x%7 != 0,s)
```

first element of the list s is always a prime

evaluation of expressions with eval()

The `eval()` executes an expression string.

```
def make_fun():
    "user given expression becomes a function"
    expr = input('give an expression in x : ')
    return lambda x : eval(expr)

FUN = make_fun()
ARG = float(input('give a value : '))
VAL = FUN(ARG)
print('the expression evaluated at %f' % ARG)
print('gives %f ' % VAL)
```

illustration of evalshow.py (delayed evaluation)

Running evalshow.py at the command prompt `$`:

```
$ python evalshow.py
give an expression in x : 2*x**6 - x + 9.9
give a value : -0.4523
the expression evaluated at -0.452300
gives 10.369423
$
```

With `eval()` we delay the evaluation of the expression entered by the user,
till a value for the variable is provided.

the apply() function

Syntax:

```
apply ( < function name > , < arguments > )
```

Although `y = apply(f,x)` equals `y = f(x)`,
the `apply` is useful when not only the arguments,
but also the function is only known at run time.

Typical example: a simple calculator.

```
$ python calculator.py
give first operand : 3
give second operand : 4
operator ? (+, -, *) *
3 * 4 = 12
```

the program calculator.py (apply() avoids if else elif)

```
from operator import add, sub, mul
OPS = { '+':add, '-':sub, '*': mul }
A = int(input('give first operand : '))
B = int(input('give second operand : '))
ACT = input('operator ? (+, -, *) ')
C = apply(OPS[ACT], (A, B))
print '%d %s %d = %d' % (A, ACT, B, C)
```

Monte Carlo without Loops

a functional implementation

Recall the Monte Carlo method to estimate π:

1. generate n points P in $[0, 1] \times [0, 1]$
2. $m := \{ (x, y) \in P : x^2 + y^2 \leq 1 \}$
3. the estimate is then $4 \times m/n$

Main ingredients in *a functional implementation*:

1. `U = lambda i:  random.uniform(0, 1)`
2. map `U` on `range(0, n)` twice, for `x` and `y`: `X`, `Y`
3. `zip(X, Y)` can with `list()` be a list of tuples
4. `T = lambda (x, y):  x**2 + y**2 <= 1`
5. `F = filter(T, Z)`

without explicit loops!

estimating π, Monte Carlo without loops

```
from random import uniform
N = int(input('Give number of samples : '))
R = range(0, N)
U = lambda i: uniform(0, 1)
X = map(U, R)
Y = map(U, R)
Z = zip(X, Y)
T = lambda (x, y): x**2 + y**2 <= 1
F = filter(T, Z)
P = 4.0*len(F)/N
print 'estimate for Pi : %f' % P
```

lambda forms
list comprehensions

1. Guessing Secrets
   - functions returning functions
   - oracles and trapdoor functions

2. anonymous functions
   - lambda forms
   - map(), reduce(), filter(), eval(), and apply()
   - estimating π with list comprehensions

3. List Comprehensions
   - algorithms and data structures
   - sequences, dictionaries, lists

4. Summary + Assignments

list comprehensions

```
>>> L = range(10)
>>> L
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> [x**2 for x in L]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> f = lambda x: x**2
```

Then the typical list comprehension works as

```
>>> [f(x) for x in L]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

making the product of two lists

In the product of two lists `A` and `B`
we take every `x` in `A` and put `x` in a tuple with every other `y` of `B`.

```
>>> A = range(3); A
[0, 1, 2]
>>> B = range(4,6); B
[4, 5]
>>> Z = [(x,y) for x in A for y in B]
>>> Z
[(0, 4), (0, 5), (1, 4), (1, 5), (2, 4), (2, 5)]
```

This is different from the `zip` of two lists:

```
>>> C = range(4,7)
>>> C
[4, 5, 6]
>>> Z = [(A[k], C[k]) for k in range(len(A))]
>>> Z
[(0, 4), (1, 5), (2, 6)]
```

estimating π with list comprehensions

```
import random
N = int(input('Give number of samples : '))
X = [random.uniform(0, 1) for i in range(N)]
Y = [random.uniform(0, 1) for i in range(N)]
Z = [(X[k], Y[k]) for k in range(N)]
T = lambda (x, y): x**2 + y**2 <= 1
R = [T(z) for z in Z]
P = 4.0*sum(R)/len(R)
print 'estimate for Pi : %f' % P
```

lambda forms
list comprehensions

Algorithms and Data Structures

a summary of Python in the small

Niklaus Wirth: programs = algorithms + data structures

Three basic control structures in any algorithm:

1. sequence of statements
2. conditional statement: if else
3. iteration: while and for loop

For every control structure,
we have a matching data structure:

	control structures	data structures
1	sequence	tuple
2	if else	dictionary
3	while / for	list

lambda forms
list comprehensions

programs are data transformations

All data are sequences of bits, or bit tuples.

Swapping values:

```
>>> a = 1
>>> b = 2
>>> (b,a) = (a,b)
>>> b
1
>>> a
2
```

Functions take sequences of arguments on input
and return sequences on output.

storing conditions: dictionaries and if else statements

We can represent an `if else` statement

```
>>> import time
>>> hour = time.localtime()[3]
>>> if hour < 12:
...     print 'good morning'
... else:
...     print 'good afternoon'
...
good afternoon
```

via a dictionary:

```
>>> d = { True:'good morning',
... False : 'good afternoon'}
>>> d[hour<12]
'good afternoon'
```

loops and lists: storing the results of a for loop

Printing all lower case characters:

```
>>> for i in range(ord('a'),ord('z')):
...     print chr(i)
```

A list of all lower case characters:

```
>>> L = range(ord('a'),ord('z'))
>>> map(chr,L)
```

`map()` returns a list of the results
of applying a function to a sequence of arguments.

The `while` statement combines `for` with `if else`:
conditional iteration.

list comprehensions: defining lists in a short way

Instead of `map()`, `filter()`, etc... (eventually with `lambda` functions), *list comprehensions* provide a shorter way to create lists:

To sample integer points on the parabola $y = x^2$:

```
>>> [(x,x**2) for x in range(0,3)]
[(0, 0), (1, 1), (2, 4)]
```

Generating three random numbers:

```
>>> from random import uniform
>>> L = [uniform(0,1) for i in range(0,3)]
>>> [ '%.3f' % x for x in L]
['0.843', '0.308', '0.272']
```

Summary + Assignments

Background reading for this lecture:

- pages 181-182 in *Python Programming in Context*,
- pages 256-258 in *Computer Science, an overview*.

Assignments:

1. Generate the list [(1,1),(1,2),(1,3),(1,4), .. ,(1,n)], for any given n. Use this list then to create all fractions $1.0/k$, for k from 1 to n. Finally, use `round()` to round all fractions to two decimal places.
2. Approximate the exponential function as $\displaystyle\sum_{k=0}^{n} \frac{x^k}{k!}$.
   Write a Python program using `map()` and `reduce()`, to evaluate this approximation for given x and n.
3. Use list comprehensions to generate points (x, y) uniformly distributed on the circle: $x^2 + y^2 = 1$.
   (For some angle t: $x = \cos(t)$, $y = \sin(t)$.)