Outline

1. Histograms
   - tallying the votes
   - global and local variables
   - call by value or call by reference
2. Arguments of Functions
   - of variable length
   - using keywords for optional arguments
3. Functions using Functions
   - the trapezoidal rule
4. Functional Programming
5. Summary + Assignments

MCS 260 Lecture 14
Introduction to Computer Science
Jan Verschelde, 12 February 2016

Histograms

interpreting results of a simulation

How do probability distributions arise in applications?

Run a simulation and tally outcomes into separate bins.

Check whether a coin is fair:

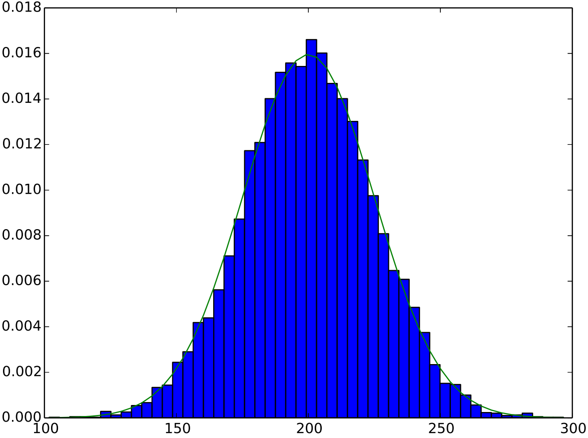
1. do a large number of coin tosses,
2. count number of heads and tails,
3. if unequal #heads and #tails, suspect unfair.

Raising the number of tosses will increase confidence.

This coin toss problem illustrates how to check whether data is uniformly distributed.

Histograms with `matplotlib`

On data randomly generated from a normal distribution, with `matplotlib` (requires `numpy`) we can plot:



global & local variables
arguments of functions

1. Histograms
   - tallying the votes
   - global and local variables
   - call by value or call by reference
2. Arguments of Functions
   - of variable length
   - using keywords for optional arguments
3. Functions using Functions
   - the trapezoidal rule
4. Functional Programming
5. Summary + Assignments

tallying votes

tossing votes as coins

Problem: make a machine to count votes.

Open democratic voting protocol (Yes or No):

1. machine says name of each member
2. upon hearing name, member says Yes or No
3. machine updates tally of Yes and No votes
4. at end of vote, program shows tally

Observe: this is a variant of the coin toss problem.

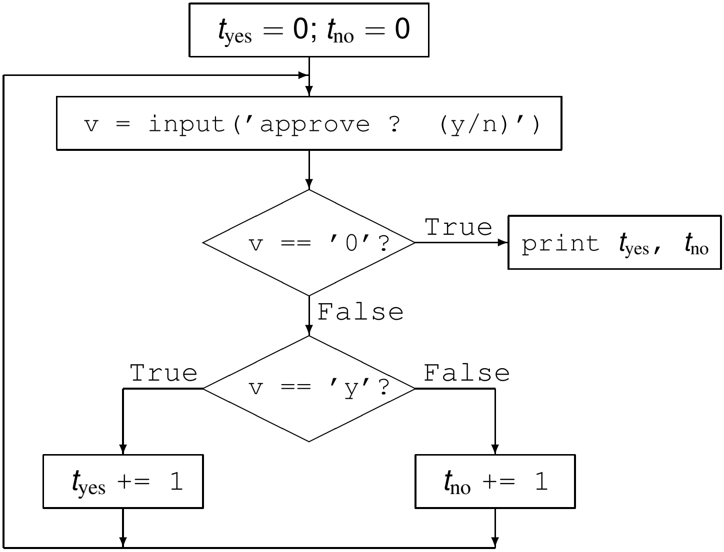
voting in action

Running the program `votes.py` at the prompt `$`:

```
$ python votes.py
Vote yes or no, 0 to stop
approve ? (y/n) y
Vote yes or no, 0 to stop
approve ? (y/n) y
Vote yes or no, 0 to stop
approve ? (y/n) n
Vote yes or no, 0 to stop
approve ? (y/n) n
Vote yes or no, 0 to stop
approve ? (y/n) y
Vote yes or no, 0 to stop
approve ? (y/n) 0
Tally of votes : [2, 3]
```

Flowchart

of the voting machine



global & local variables
arguments of functions

1. Histograms
   - tallying the votes
   - global and local variables
   - call by value or call by reference
2. Arguments of Functions
   - of variable length
   - using keywords for optional arguments
3. Functions using Functions
   - the trapezoidal rule
4. Functional Programming
5. Summary + Assignments

global and local variables

hierarchy imposed on data

In top down design we distinguish between

- functions that focus on one particular task
- the main program that calls the functions

Also the data fits into two categories:

- variables inside a function are *local*
- data managed by the main program is *global*

Example, in the voting machine:

- the variable to store the answer will be local
- the tally of the votes is global

an example of a global variable

```
VALUE = 2014   # our global variable

def update(formalv):
    """
    shows value of formal v
    and prompts for new v
    """
    print('v = ', formalv)
    vraw = input('Give new value : ')
    newv = int(vraw)
    return newv

while True:
    VALUE = update(VALUE) # do not forget ()
    ANS = input('continue ? (y/n) ')
    if ANS != 'y':
        break
```

Python functions are functions

A function f in the proper mathematical sense, called like $y = f(x)$, does not change the argument x of the function.

Updating the tally `t` with vote `v` with the function `update(t, v)`, called as `t = update(t, v)`, where `t = (tno, tyes)`.

The function `update` will

1. check the value of the vote `v`
2. create a new tuple with updated values
3. return the new tuple

The caller of `update(t, v)` assigns the updated values to `tno` and `tyes`.

Some terminology:

- call by value: with tuples (*immutable*)
- call by reference: with lists (*variable*).

global & local variables
arguments of functions

1. Histograms
   - tallying the votes
   - global and local variables
   - call by value or call by reference
2. Arguments of Functions
   - of variable length
   - using keywords for optional arguments
3. Functions using Functions
   - the trapezoidal rule
4. Functional Programming
5. Summary + Assignments

the skeleton of votes.py – top down design

```
TNO = 0    # tno counts no votes
TYES = 0   # tyes counts yes votes

def poll():
    "asks whether approve or not"

def update(tally, vote):
    "updates tally with vote"

while True:
    VOTE = poll()
    if VOTE == '0':
        break
    (TNO, TYES) = update((TNO, TYES), VOTE)
print 'Tally of votes :', (TNO, TYES)
```

the functions

```
def poll():
    "asks whether approve or not"
    print 'Vote yes or no, 0 to stop'
    answer = raw_input('approve ? (y/n) ')
    return answer
```

answer is a local variable in poll

```
def update(tally, vote):
    "updates tally with vote"
    if vote == 'y':
        return (tally[0], tally[1]+1)
    elif vote == 'n':
        return (tally[0]+1, tally[1])
```

we do *not* assign to t or its components

assignments and lists – using the side effects

```
>>> yes = 3; no = 2
>>> tally = [no,yes]
>>> t = tally
>>> tally
[2, 3]
>>> t
[2, 3]
>>> t[1] = t[1]+1
>>> t
[2, 4]
>>> tally
[2, 4]
```

We do not assign to `tally`, as `L` refers to the *same* list as `tally`, assigning to a component of `t` also changes `tally`.

call by reference

The second version of tallying votes with same `poll()`:

```
TALLY = [0, 0] # TALLY[0] counts no votes
               # TALLY[1] counts yes votes

def update(tally, vote):
    "updates tally with vote"
    if vote == 'y':
        tally[1] += 1
    elif vote == 'n':
        tally[0] += 1

while True:
    VOTE = poll()
    if VOTE == '0':
        break
    update(TALLY, VOTE)
print 'Tally of votes :', TALLY
```

global & local variables
arguments of functions

Arguments of Variable Length

Consider the area computation of a square or rectangle.

The dimensions of a rectangle are length and width,
but for a square we only need the length.

$\rightarrow$ functions whose number of arguments is variable.

The arguments which may or may not appear
when the function is called are collected in a tuple.

Python syntax:

```
def < name > ( < args > , * < tuple > ) :
```

The name of the `tuple` must

- appear after all other arguments `args`,
- and be preceded by `*`.

area of square or rectangle

```
def area ( length, *width ):
    "returns area of rectangle"
    if len(width) == 0:          # square
        return length**2
    else:                        # rectangle
        return length*width[0]
```

Observe the different meanings of * !

```
print 'area of square or rectangle'
WLEN = raw_input('give length : ')
LEN = float(WLEN)
WRAW = raw_input('give width : ')
WID = float(WRAW)
if WID == 0:
    AREA = area(LEN)
else:
    AREA = area(LEN, WID)
```

global & local variables arguments of functions

using keywords

If arguments are optional, then we may identify the extra arguments of a function with keywords.

Instead of `a = area(L,W)`
we require `a = area(L, width=W)`.

Python syntax:

```
def < f > ( < a > , * < t > , ** < dict > ) :
```

The name of the dictionary `dict` must

- appear at the very end of the arguments,
- and be preceded by `**`.

optional arguments

```
def area ( length, **width ):
    "returns area of rectangle"
    if len(width) == 0:          # square
        return length**2
    else:                        # rectangle
        result = length
        for each in width:
            result *= width[each]
        return result
```

observe the access to the dictionary ...

```
# input of LEN and WID omitted
if WID == 0:
    AREA = area(LEN)
else:
    AREA = area(LEN, width=WID)
print 'the area is', AREA
```

Calling `area(L,W)` no longer possible.

global & local variables
arguments of functions

1. Histograms
   - tallying the votes
   - global and local variables
   - call by value or call by reference
2. Arguments of Functions
   - of variable length
   - using keywords for optional arguments
3. Functions using Functions
   - the trapezoidal rule
4. Functional Programming
5. Summary + Assignments

functions using functions

To approximate the integral of a function $f(x)$ over $[a, b]$,
the trapezoidal rule is

$$\int_a^b f(x)dx \approx \frac{1}{2}(f(a) + f(b))(b - a).$$

Geometrically, we approximate the area under $f(x)$
for $x \in [a, b]$ by the area of a trapezium,
with base $[a, b]$ and heights $f(a)$ and $f(b)$.

A function as argument of a Python function, template:

```
def < rule > ( < f > , < a > , < b > ) :
    "integrate function f over [a,b]"
```

the trapezoidal rule in Python

```
def traprule(fun, start, stop):
    "trapezoidal rule for f(x) over [start, stop]"
    return (stop-start)*(fun(start) + fun(stop))/2

from math import exp
RESULT = 'integrating exp() over '
print RESULT + '[a,b]'
ARAW = raw_input('give a : ')
A = float(ARAW)
BRAW = raw_input('give b : ')
B = float(BRAW)
APPROX = traprule(exp, A, B)
print RESULT + '[%.1E, %.1E] : ' % (A, B)
print 'the approximation : %.15E' % APPROX
EXACT = exp(B) - exp(A)
print '  the exact value : %.15E' % EXACT
```

running traprule.py

Running `traprule.py` at the prompt $

```
$ python traprule.py
integrating exp() over [a,b]
give a : 0
give b : 1
integrating exp() over [0.0E+00,1.0E+00] :
the approximation : 1.85914091422952E+00
  the exact value : 1.71828182845904E+00
```

Using functions as parameters to other functions allows to write more *generic* functions.

Example: finding the minimum or maximum in a list,
use comparison function ($>$ or $<$) as argument.

functional programming

Lisp

Early high level programming languages like `C` are heavily influenced by *the Von Neumann architecture*.

What this means is that the programmer is aware of the internal workings of the computer. For example, a skilled C programmer knows the distinction between the contents of a memory cell and its address.

Advantages and criticisms:

+ the programmer has great power and flexibility
– the description of algorithms is independent of computers

Except for recursion, we know already enough of Python to apply functional programming.

Summary + Assignments

Background reading for this lecture:

- pages 146-150 in *Python Programming in Context*,
- pages 273-279 in *Computer Science, an overview*.

Assignments:

1. Simulate a coin toss in a Python program, applying `random.randint(0,1)` at least a thousand times. Count the number of 0s and 1s. Is Python's coin fair?
2. Modify the vote tally to include abstain votes.
3. Extend the area function into the volume computation of a cube, or general parallelepiped. For a cube, only one parameter will be given, otherwise, the user must specify length, width, and height.