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, 26 September 2008
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
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.
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
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
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.
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.
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 in Maple

Given any list of data, \texttt{stats[statplots, histogram]} produces a plot:

\begin{verbatim}
> data := [stats[random, normald[0,1]](5000)]:
> stats[statplots, histogram](data);
\end{verbatim}
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**
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.
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.
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.
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.
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.
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]
```
Histograms
tallying the votes
global and local variables
call by value or call by reference

Arguments of Functions
of variable length
using keywords for optional arguments

Functions using Functions
the trapezoidal rule

Functional Programming

Summary + Assignments

Flowchart of the voting machine

$t_{yes} = 0; t_{no} = 0$

$v = \text{input}('\text{approve ? (y/n)}')$

$v == '0'$?

False

print $t_{yes}, t_{no}$

True

$v == 'y'$?

$v == '0'$?

$v == 'y'$?

$t_{yes} += 1$

$t_{no} += 1$
Histograms
tallying the votes
global and local
variables
call by value or call
by reference

Arguments of
Functions
of variable length
using keywords for
optional arguments

Functions
using Functions
the trapezoidal rule

Functional
Programming

Summary +
Assignments

---

Histograms
tallying the votes
global and local
variables
call by value or call
by reference

Arguments of
Functions
of variable length
using keywords for
optional arguments

Functions
using Functions
the trapezoidal rule

Functional
Programming

Summary +
Assignments

---

Flowchart
of the voting machine

```
$ t_{yes} = 0; t_{no} = 0 $

$v = \text{input('approve ? (y/n)')} $

$v == '0'? True$

---

$ t_{yes} += 1$

$ t_{no} += 1$

---

$v == 'y'? True$

---

print $t_{yes}, t_{no}$

---

Flowchart
of the voting machine

```

---

Flowchart
of the voting machine

```

---

Flowchart
of the voting machine

```
Flowchart of the voting machine

$t_{yes} = 0; t_{no} = 0$

$v = \text{input('approve ? (y/n)' )}$

$v == '0'$?

True

print $t_{yes}, t_{no}$

False

$v == 'y'$?

True $t_{yes} += 1$

False $t_{no} += 1$
Histograms

tallying the votes

global and local variables
call by value or call by reference

Arguments of Functions

of variable length
using keywords for optional arguments

Functions

using Functions
the trapezoidal rule

Functional Programming

Summary + Assignments

---

Flowchart of the voting machine

$t_{yes} = 0; t_{no} = 0$

$v = \text{input}('\text{approve} ? \ (y/n)')$

$v == '0'$?

True

print $t_{yes}, t_{no}$

False

True

$v == 'y'$?

False

$t_{yes} += 1$

$t_{no} += 1$

---
Flowchart of the voting machine

\[ t_{yes} = 0; t_{no} = 0 \]

\[ v = \text{input}('\text{approve ? (y/n)}') \]

- **v == '0'**?
  - True
    - print \( t_{yes}, t_{no} \)
  - False
    - \( v == 'y' ? \)
      - True
        - \( t_{yes} += 1 \)
      - False
        - \( t_{no} += 1 \)
Histograms
tallying the votes
global and local
variables
call by value or call
by reference

Arguments of
Functions
of variable length
using keywords for
optional arguments

Functions
using
Functions
the trapezoidal rule

Functional
Programming

Summary +
Assignments

Flowchart
of the voting machine

\[ t_{\text{yes}} = 0; \ t_{\text{no}} = 0 \]

\[ \text{v} = \text{input}(\text{‘approve ? (y/n)’}) \]

\[ \text{v} == \text{‘0’} ? \]
\[ \text{True} \]
\[ \text{print } t_{\text{yes}}, \ t_{\text{no}} \]
\[ \text{False} \]

\[ \text{v} == \text{‘y’} ? \]
\[ \text{True} \]
\[ t_{\text{yes}} += 1 \]
\[ \text{False} \]
\[ t_{\text{no}} += 1 \]
Histograms

tallying the votes

global and local
variables

call by value or call
by reference

Arguments of
Functions

of variable length

using keywords for
optional arguments

Functions
using

Functions

the trapezoidal rule

Functional
Programming

Summary +
Assignments

Flowchart
of the voting machine

$t_{yes} = 0; t_{no} = 0$

$v = \text{input}(\text{‘approve ? (y/n)’})$

$v == '0'$?

True

print $t_{yes}, t_{no}$

False

$v == 'y'$?

True

$t_{yes} += 1$

False

$t_{no} += 1$
Flowchart of the voting machine

$t_{yes} = 0; t_{no} = 0$

$v = \text{input}('approve ? (y/n)')$

$v == '0'$?

$\text{True}$

print $t_{yes}, t_{no}$

$\text{False}$

$v == 'y'$?

$\text{True}$

t$_{yes} += 1$

$\text{False}$

t$_{no} += 1$
Histograms
tallying the votes
global and local variables
call by value or call by reference

Arguments of Functions
of variable length
using keywords for optional arguments

Functions using Functions
the trapezoidal rule

Functional Programming

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
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
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
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 \textit{local}
- data managed by the main program is \textit{global}

Example, in the voting machine:

- the variable to store the answer will be local
- the tally of the votes is global
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 $\text{update}(t, v)$, called as $t = \text{update}(t, v)$, where $t = (\text{tno}, \text{tyes})$.

The function $\text{update}$ 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 $\text{update}(t, v)$ assigns the updated values to $\text{tno}$ and $\text{tyes}$.

Some terminology:

- call by value: with tuples (immutable)
- call by reference: with lists (variable).
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 \( \text{update}(t,v) \), called as \( t = \text{update}(t,v) \), where \( t = (tno,tyes) \).

The function \( \text{update} \) 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 \( \text{update}(t,v) \) assigns the updated values to \( tno \) and \( tyes \).

Some terminology:

- call by value: with tuples (immutable)
- call by reference: with lists (variable).
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 $\text{update}(t,v)$, called as $t = \text{update}(t,v)$, where $t = (\text{tno}, \text{tyes})$.

The function $\text{update}$ 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 $\text{update}(t,v)$ assigns the updated values to $\text{tno}$ and $\text{tyes}$.

Some terminology:

- call by value: with tuples (immutable)
- call by reference: with lists (variable)
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 $\text{update}(t, v)$, called as $t = \text{update}(t, v)$, where $t = (\text{tno}, \text{tyes})$.

The function $\text{update}$ 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 $\text{update}(t, v)$ assigns the updated values to $\text{tno}$ and $\text{tyes}$.

Some terminology:

- call by value: with tuples (immutable)
- call by reference: with lists (variable).
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 $\text{update}(t, v)$, called as $t = \text{update}(t, v)$, where $t = (\text{tno}, \text{tyes})$.

The function $\text{update}$ 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 $\text{update}(t, v)$ assigns the updated values to $\text{tno}$ and $\text{tyes}$.

Some terminology:

- call by value: with tuples (immutable)
- call by reference: with lists (variable).
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 $\text{update}(t, v)$, called as $t = \text{update}(t, v)$, where $t = (\text{tno}, \text{tyes})$.

The function $\text{update}$ 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 $\text{update}(t, v)$ assigns the updated values to $\text{tno}$ and $\text{tyes}$.

Some terminology:

- call by value: with tuples (immutable)
- call by reference: with lists (variable)
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 \( \text{update}(t,v) \), called as \( t = \text{update}(t,v) \), where \( t = (tno,tyes) \).

The function \( \text{update} \) 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 \( \text{update}(t,v) \) assigns the updated values to \( tno \) and \( tyes \).

Some terminology:

- call by value: with tuples (immutable)
- call by reference: with lists (variable).
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 $\text{update}(t, v)$, called as $t = \text{update}(t, v)$, where $t = (\text{tno}, \text{tyes})$.

The function $\text{update}$ 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 $\text{update}(t, v)$ assigns the updated values to $\text{tno}$ and $\text{tyes}$.

Some terminology:

- call by value: with tuples (immutable)
- call by reference: with lists (variable)
Histograms
tallying the votes
global and local variables
call by value or call by reference

Arguments of Functions
of variable length
using keywords for optional arguments

Functions using Functions
the trapezoidal rule

Functional Programming

Summary + Assignments
The Skeleton of votes.py

top down design in words

# L-14 MCS 260 Fri 26 Sep 2008 histograms
#
# A voting machine tallies votes.
#
tno = 0  # tno counts no votes
ty = 0    # tyes counts yes votes

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

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

while True:  # main program
    v = poll()
    if v == '0': break
    (t, y) = update((t, y), v)
print 'Tally of votes:', (t, y)
The Skeleton of votes.py

top down design in words

# L-14 MCS 260 Fri 26 Sep 2008  histograms
#
# A voting machine tallies votes.
#
tno = 0  # tno counts no votes
tyes = 0  # tyes counts yes votes

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

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

while True:    # main program
    v = poll()
    if v == '0': break
    (tno,tyes) = update((tno,tyes),v)
print 'Tally of votes:', (tno,tyes)
The Skeleton of votes.py

top down design in words

# L-14 MCS 260 Fri 26 Sep 2008  histograms
#
# A voting machine tallies votes.
#
tno = 0  # tno counts no votes
tyes = 0  # yes counts yes votes

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

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

while True:  # main program
    v = poll()
    if v == '0': break
    (tno,tyes) = update((tno,tyes),v)
print 'Tally of votes :', (tno,tyes)
# L-14 MCS 260 Fri 26 Sep 2008 histograms
#
# A voting machine tallies votes.
#
tno = 0  # tno counts no votes
tyes = 0  # tyes counts yes votes

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

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

while True:  # main program
    v = poll()
    if v == '0': break
    (tno,tyes) = update((tno,tyes),v)
print 'Tally of votes :', (tno,tyes)
The Functions

poll() and update(t,vote)

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(t,vote):
    "updates tally t with vote"
    if vote == 'y':
        return (t[0],t[1]+1)
    elif vote == 'n':
        return (t[0]+1,t[1])

we do not assign to t or its components
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(t,vote):
    "updates tally t with vote"
    if vote == 'y':
        return (t[0],t[1]+1)
    elif vote == 'n':
        return (t[0]+1,t[1])

we do *not* assign to t or its components
The Functions

poll() and update(t,vote)

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(t,vote):
    "updates tally t with vote"
    if vote == 'y':
        return (t[0],t[1]+1)
    elif vote == 'n':
        return (t[0]+1,t[1])

we do *not* assign to t or its components
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(t,vote):
    "updates tally t with vote"
    if vote == 'y':
        return (t[0],t[1]+1)
    elif vote == 'n':
        return (t[0]+1,t[1])

we do *not* assign to t or its components
Assignments and Lists

using the side effects

```python
>>> 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`.
Assignments and Lists  
using the side effects

```python
>>> 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 `t` refers to the *same* list as `tally`, assigning to a component of `t` also changes `tally`.

Histories  
tallying the votes  
global and local variables  
call by value or call by reference

Arguments of Functions  
of variable length  
using keywords for optional arguments

Functions  
the trapezoidal rule

Functional Programming

Summary + Assignments
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`. 
### Assignments and Lists

using the side effects

```python
tally = [no, yes] 
t = tally
```

```python
>>> yes = 3; no = 2
>>> tally = [no, yes]
>>> t = tally
[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

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

tally = [0,0]  # tally[0] counts no votes
    # tally[1] counts yes votes

```python
def update(t, vote):
    "updates tally t with vote"
    if vote == 'y':
        t[1] += 1
    elif vote == 'n':
        t[0] += 1

while True:
    v = poll()
    if v == '0': break
    update(tally, v)
print 'Tally of votes :', tally
```
Call by Reference

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

tally = [0, 0] # tally[0] counts no votes
    # tally[1] counts yes votes

def update(t, vote):
    "updates tally t with vote"
    if vote == 'y':
        t[1] += 1
    elif vote == 'n':
        t[0] += 1

while True:
    v = poll()
    if v == '0': break
    update(tally, v)
print 'Tally of votes: ', tally
Histograms tallying the votes
global and local variables
call by value or call by reference
Arguments of Functions
of variable length
using keywords for optional arguments
Functions using Functions
the trapezoidal rule
Functional Programming
Summary + Assignments

Call by Reference

second version of tallying votes with same poll()

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

def update(t,vote):
    "updates tally t with vote"
    if vote == 'y':
        t[1] += 1
    elif vote == 'n':
        t[0] += 1

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

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
Arguments of Variable Length
defining more general functions

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.
→ 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:

```python
def < name > ( < args > , * < tuple > ) :
```
The name of the tuple must
- appear after all other arguments `args`,
- and be preceded by `*`. 
Arguments of Variable Length

defining more general functions

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.

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 *.
Arguments of Variable Length

defining more general functions

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.

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 *.
Arguments of Variable Length
defining more general functions

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.

→ 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:

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

The name of the `tuple` must

• appear after all other arguments `args`,

• and be preceded by `*`.  

• using keywords for optional arguments

Arguments of variable length

Functions of variable length

Functions using the trapezoidal rule

Functional Programming

Summary + Assignments
Arguments of Variable Length

defining more general functions

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.

to 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 *.
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. → 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:

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

The name of the `tuple` must
- appear after all other arguments `args`,
- and be preceded by `*`. 
Arguments of Variable Length

defining more general functions

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.

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

```python
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'
L = input('give length : ')
W = input('give width : ')
if W == 0:
a = area(L)
else:
a = area(L,W)
print 'the area is', a
```
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'
L = input('give length : ')
W = input('give width : ')
if W == 0:
    a = area(L)
else:
    a = area(L,W)
print 'the area is', a
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
If arguments are optional, then we may identify the extra arguments of a function with keywords.

Instead of \( a = \text{area}(L,W) \)
we require \( a = \text{area}(L, \text{width}=W) \).

Python syntax:

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

The name of the dictionary `dict` must

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

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

Instead of \(a = \text{area}(L, W)\)

we require \(a = \text{area}(L, \text{width}=W)\).

Python syntax:

\[
\text{def } < f > ( < a > , * < t > , ** < dict > ) :
\]

The name of the dictionary \text{dict} must

- appear at the very end of the arguments,
- and be preceded by **.
Using Keywords for optional arguments

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

Instead of \( a = \text{area}(L,W) \)
we require \( a = \text{area}(L,\text{width}=W) \).

Python syntax:

```python
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
        a = length
        for each in width:
            a *= width[each]
        return a

observe the access to the dictionary ...
# input of L and W omitted
if W == 0:
    a = area(L)
else:
    a = area(L,width=W)
print 'the area is', a

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
the trapezoidal rule

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:

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

```python
def < rule > ( < f > , < a > , < b > ) :
    "integrate function f over [a,b]"
```
Functions using Functions
the trapezoidal rule

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:

```python
def < rule > ( < f > , < a > , < b > ) :
    "integrate function f over [a,b]"
```
The Trapezoidal Rule in Python

def traprule(f,a,b):
    "trapezoidal rule for f(x) over [a,b]"
    return (b-a)*(f(a) + f(b))/2

import math
s = 'integrating exp() over ' 
print s + '[a,b]'
a = input('give a : ')
b = input('give b : ')
y = traprule(math.exp,a,b)
print s + '[%.1E,%.1E] : ' % (a,b)
print 'the approximation : %.15E' % y
e = math.exp(b) - math.exp(a)
print ' the exact value : %.15E' % e
The Trapezoidal Rule in Python

def traprule(f, a, b):
    "trapezoidal rule for f(x) over [a,b]"
    return (b-a)*(f(a) + f(b))/2

import math
s = 'integrating exp() over ' 
print s + '[a,b]'

a = input('give a : ')
b = input('give b : ')
y = traprule(math.exp, a, b)

print s + '[%.1E,%.1E] : ' % (a,b)
print 'the approximation : %.15E' % y

e = math.exp(b) - math.exp(a)
print ' the exact value : %.15E' % e
The Trapezoidal Rule in Python

def traprule(f, a, b):
    "trapezoidal rule for f(x) over [a,b]"
    return (b-a)*(f(a) + f(b))/2

import math
s = 'integrating exp() over '
print s + '[a,b]'
a = input('give a : ')
b = input('give b : ')
y = traprule(math.exp, a, b)
print s + '[%.1E,%.1E] : ' % (a,b)
print 'the approximation : %.15E' % y
e = math.exp(b) - math.exp(a)
print 'the exact value : %.15E' % e
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 : 1.859140914229523E+00
the exact value : 1.718281828459045E+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.
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 : 1.859140914229523E+00
  the exact value : 1.718281828459045E+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.
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 : 1.859140914229523E+00
the exact value : 1.718281828459045E+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

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.
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.
Functional Programming

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.
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.
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.
Functional Programming

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.
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.
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:

- chapter 7 in *Python Power!*
- 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.

Homework will be collected on Wednesday 1 October.