Outline

MCS 260 Lecture 13
Introduction to Computer Science
Jan Verschelde, 10 February 2016

top down design
functions in Python

What If Scenarios

should we buy another printer?

Estimate wait times for printer jobs to finish.

Problem: printer shared by several users.

- the printer queues jobs along FIFO protocol
- arrival times are uniformly distributed
- length of the jobs is normally distributed

Given n jobs arriving uniformly in time interval $[0, T]$,
with average length μ and standard deviation σ;

what is the average wait time?
And what is the standard deviation of the wait times?

top down design
functions in Python

top down design

divide work in separate tasks and conquer

Observe the logical division of actions:
submitting jobs and printing jobs are separate tasks.

Key point: separate making of jobs from processing jobs.

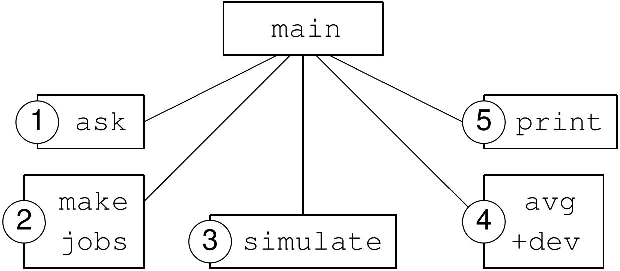
Five tasks:

1. ask the user for the parameters of the simulation
2. make jobs, generate arrival and processing times
3. simulate printing and compute wait times
4. compute average wait time and standard deviation
5. print the results

There will be one main program and five functions,
one function for each task.

tree structure

a hierarchy of tasks



Plus one utility function to apply format `'%.2f'` to lists of floats.

data flow

Input/Output descriptions for the five tasks:

1. ask the user for the parameters of the simulation
   output: (n, T, μ, σ)
2. make jobs, generate arrival and processing times
   input: (n, T, μ, σ)
   output: lists A and T with times
3. simulate printing and compute wait times
   input: lists A and T with times
   output: list of waiting times W
4. compute average wait time and standard deviation
   input: list of waiting times W
   output: average a and deviation d
5. print the results
   input: average a and deviation d

mathematical functions

arguments are variables and parameters

Suppose we want to evaluate

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

the probability density function of a normal distribution.

The symbols in the formula are

- x is the name of the variable
- μ is the mean of the distribution
- σ is the standard deviation from the mean
- $\sqrt{\ }$ is the square root function
- π is the mathematical constant π
- e is the exponential function

top down design
functions in Python

definition and arguments

The general syntax of a function definition is

```
def < function name > ( < arguments > ):
     < function body >
```

All statements in the body must be indented!

For the arguments, we distinguish between

1. required arguments that always must be given
2. optional arguments have default values

The normal probability density function `npdf`:

```
def npdf(arg, mean=0, sigma=1):
```

has one required argument: `arg` and
two optional arguments: mean `mean` (with default value 0),
standard deviation `sigma` (with default value 1).

top down design
functions in Python

the documentation string

The body of the function starts typically with a *documentation string*, to document the function: what it does, its arguments.

```
def npdf(arg, mean=0, sigma=1):
    "normal probability density function"
```

In a Python shell:

```
>>> import npdf
>>> help(npdf)

NAME
    npdf - Same name for function and module.

FILE
    /Users/jan/Courses/MCS260/Spring15/Lec13/npdf.py

FUNCTIONS
    npdf(arg, mean=0, sigma=1)
        normal probability density function
```

the body of a function

The square root and exponential function are available as `math.sqrt` and `math.exp` in the math module.

```
def npdf(arg, mean=0, sigma=1):
    "normal probability density function"
    import math
    result = math.exp(-(arg - mean)**2/(2*sigma**2))
    result = result/(sigma*math.sqrt(2*math.pi))
    return result
```

Because of the `return`, we may call the function
– if saved in the file `npdf.py` – as:

```
>>> import npdf
>>> y = npdf.npdf(2)
>>> npdf.npdf(2,2.3,0.1)
0.044318484119380351
```

using keywords as arguments

When there are multiple arguments, confusing the order in which the arguments must be provided leads to errors.

```
>>> import npdf
>>> npdf.npdf(2,2.3,0.1)
0.044318484119380351
```

This is equivalent to

```
>>> npdf.npdf(mu=2.3, sigma=0.1, arg=2)
0.044318484119380351
```

The names `mu` and `sigma` are *formal parameters*.
They are not variables like `result` inside the definition of `npdf`.

formal and actual parameters

```
def f(x):
    return x**2

a = 2
b = f(a)
```

- The argument `x` of `f` is a formal parameter.
- At the call `b = f(a)`,
  the *formal parameter* with name `x` refers to the object which is referred to by the *actual parameter* with name `a`.
- The `return` statement assigns the object that holds the value `x**2` to the variable `b`.
- Note that `a` and `b` are *outside the scope* of `f`:
  the definition of `f` cannot use `a` nor `b`.
  Moreover: `x` does not exist outside the definition of `f`.

execution of function call

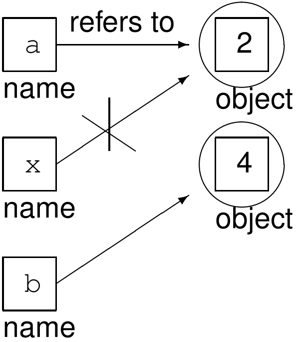
```
def f(x):
    return x**2

a = 2
b = f(a)
```

after `a = 2`:

calling `f(a)`:

after `b = f(a)`:



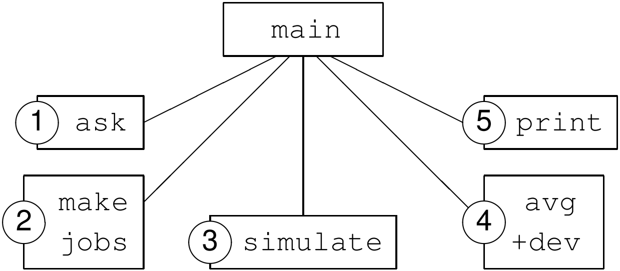
use in a program

```
def npdf(arg, mean=0, sigma=1):
    "normal probability density function"
    import math
    result = math.exp(-(arg - mean)**2/(2*sigma**2))
    result = result/(sigma*math.sqrt(2*math.pi))
    return result

ARG = float(input('give x : '))
print('f(', ARG, ') = ', npdf(ARG))
MEAN = float(input('give mean : '))
SIGMA = float(input('give standard deviation : '))
print('for mu = ', MEAN, 'and sigma = ', SIGMA)
print('f(', ARG, ') = ', \
    npdf(mean=MEAN, sigma=SIGMA, arg=ARG))
```

implementing the simulation

Recall the tree of functions:



Plus one utility function to apply format `'%.2f'` to lists of floats.

a session with the program

Let us print 5 jobs within an hour,
mean is 0.3 and standard deviation is 0.1.

Running `simuwait.py`

```
$ python simuwait.py
number of jobs : 5
length of time : 1
 mean time/job : 0.3
 deviation/job : 0.1
arrivals : [ 0.66, 0.76, 0.77, 0.82, 0.85 ]
job times : [ 0.17, 0.28, 0.31, 0.31, 0.29 ]
wait times : [ 0.00, 0.07, 0.34, 0.60, 0.88 ]
average wait : 0.38
   deviation : 0.73
```

top down design
functions in Python

the main program – names of variables and functions

The parameters are `dim`, `dur`, `mean`, and `sigma`. The lists `arv`, `prc`, and `wait` collect the data. Results are in `avg` and `dev`.
Functions are `ask`, `make_jobs`, `form`, `simulate`, and `avgdev`.

```
def main():
    """
    Simulation of waiting times.
    """
    (dim, dur, mean, sigma) = ask()
    (arv, prc) = make_jobs(dim, dur, mean, sigma)
    print(' arrivals : ' + form(arv))
    print('job times : ' + form(prc))
    wait = simulate(arv, prc)
    print('wait times : ' + form(wait))
    (avg, dev) = avgdev(wait)
    print('average wait : %.2f' % avg)
    print('   deviation : %.2f' % dev)
```

asking for input parameters – no input arguments

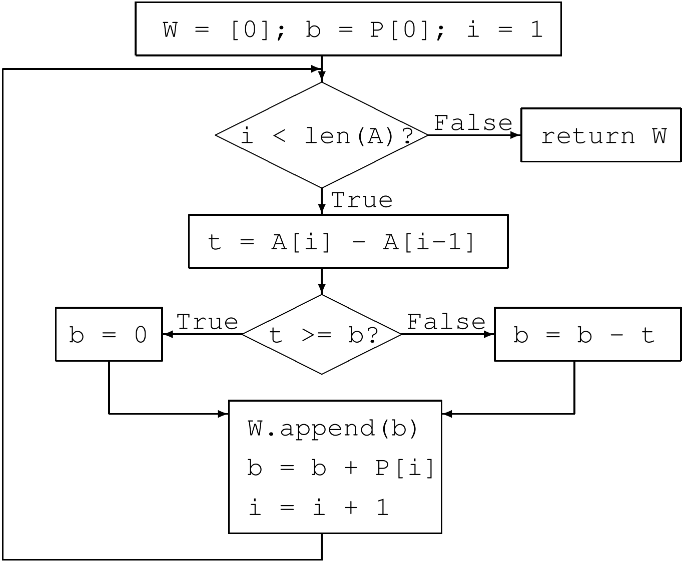
```
def ask():
    """
    Prompts the user for the parameters of the simulation:
    nbjobs : the number of jobs,
    time : the length of simulation time,
    mean : the average time per job,
    sigma : the deviation in procesing times.
    Returns the tuple (nbjobs, time, mean, sigma).
    """
    nbjobs = int(input('number of jobs : '))
    time = int(nput('length of time : '))
    mean = float(input(' mean time/job : '))
    sigma = float(input(' deviation/job : '))
    return (nbjobs, time, mean, sigma)
```

calling random number generators for the job queue

```
def make_jobs(nbjobs, time, mean, sigma):
    """
    Given the parameters of the simulation,
    returns tuple of two lists with arrival
    and process times for each job.
    """
    atime = [] # arrival times of jobs
    ptime = [] # time to process each job
    import random
    for i in range(0, nbjobs):
        atime.insert(0, random.uniform(0, time))
        ptime.insert(0, random.gauss(mean, sigma))
        print('job', i, ':', atime[0], ptime[0])
    atime.sort()
    return (atime, ptime)
```

Note that arrival times must be sorted.
Rearranging processing times is not needed.

flowchart for the simulation



simulating the printing

Taking the job queue as input, we are now able to compute the wait time for each job.

```
def simulate(arr, prc):
    """
    Given a list of arrivals and process times,
    returns a list of wait times for each job.
    """
    result = [0]    # no wait for first job
    busy = prc[0]   # time busy for job
    for i in range(1, len(arr)):
        elp = arr[i] - arr[i-1] # elapsed time
        if elp >= busy:         # idle printer
            busy = 0            # no wait
        else:                   # busy printer
            busy = busy - elp   # wait
        result.append(busy)     # store wait
        busy = busy + prc[i]    # update busy
    return result
```

average and deviation

Formulas:

$$a = \sum_{i=0}^{\text{len}(W)-1} W[i] \quad d = \sqrt{\sum_{i=0}^{\text{len}(W)-1} (W[i] - a)^2}$$

```
def avgdev(wait):
    """
    Returns a tuple with the average and
    standard deviation of the numbers in wait.
    """
    from math import sqrt
    avg = sum(wait)/len(wait)
    dev = 0
    for i in range(0, len(wait)):
        dev += (wait[i] - avg)**2
    dev = sqrt(dev)
    return (avg, dev)
```

formatting lists

The formatting of a list is realized via the creation of a string.
Printing the string returned by `form` gives the desired format.

```
def form(data):
    """
    Given in data a list of floats, returns a string
    that contains the floats in %.2f format.
    """
    result = '[ ' + '%.2f' % data[0]
    for i in range(1, len(data)):
        result += ', ' + '%.2f' % data[i]
    return result + ' ]'
```

This function is called multiple times.

top down design
functions in Python

Lambda Forms

To define functions quickly:

```
>>> f = lambda x: x**2
>>> f(3)
9

>>> f
<function <lambda> at 0x6e1b0>
>>> type(f)
<type 'function'>

>>> R = list(range(2, 8))
>>> R
[2, 3, 4, 5, 6, 7]
>>> [f(x) for x in R]
[4, 9, 16, 25, 36, 49]
```

Summary + Assignments

We covered in the lecture:

- section 6.3 in *Computer Science, an overview*,
- pages 29-32 in *Python Programming in Context*.

Assignments:

1. A word is a palindrome if it reads the same backwards as forwards. Draw the flowchart for an algorithm to decide if a string is a palindrome.
2. Write a Python function for exercise 1. The function takes on input a string and returns `True` if the string is a palindrome, `False` otherwise.
3. Add print statements in the `Simulate` function to set up a table that records all values of `t` and `b`.
4. Adjust the program for multiple printers.