DOING SOMETHING USEFUL WITH PYTHON

ES 112

Brief Recap: Doing Something with the Data

- (A bit more on) Conditionals
- Iterations (`while`, `for` and `break`)

Menu for Today! Structuring Programs

- Iteration patterns
- Structuring Programs

Iteration Patterns

- enumeration
- accumulation
- Combining enumeration and accumulation
- Examples

  `count`

  `sum`

  `average`

Iteration Pattern: enumeration

```
count = 0
number = int(input(‘Gimme a number ’)
while (number >= 0):
    count = count + 1
    number = int(input(‘Gimme a number ’)
```

Iteration Pattern: accumulate

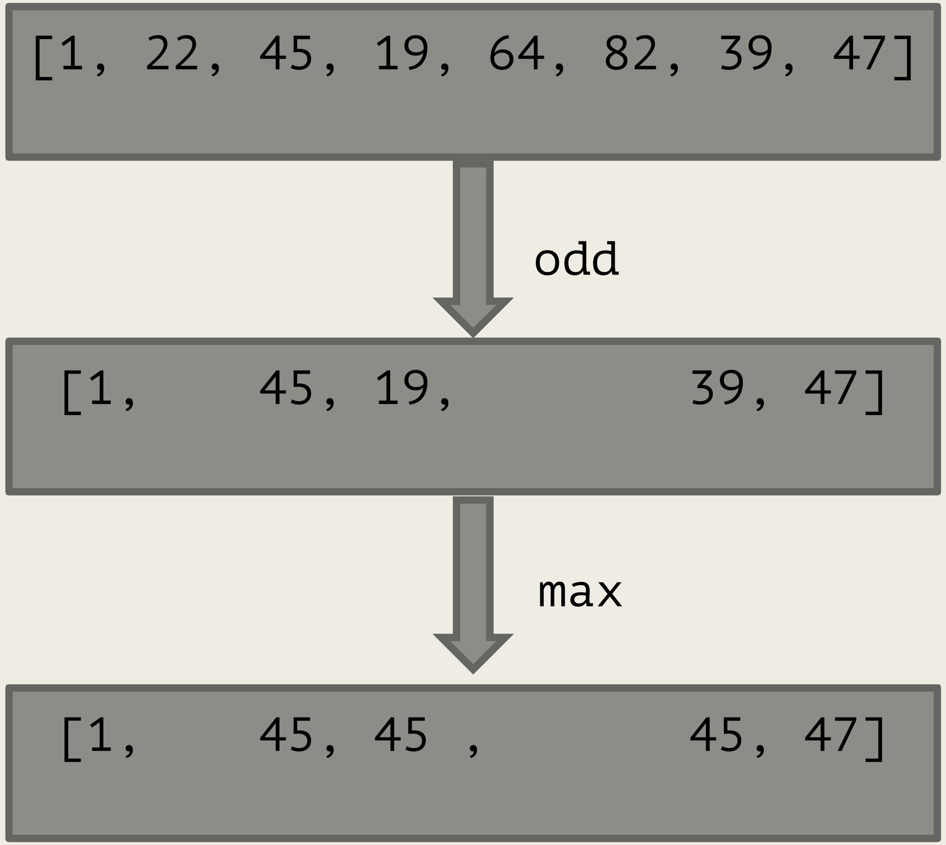
```
sum = 0
number = int(input(‘Gimme a number ’)
while (number >= 0):
    sum = sum + number
    number = int(input(‘Gimme a number ’)
```

Combining the Two Patterns

```
sum = 0
count = 0
done = False
number = int(input(‘Gimme a number ’)
while (number >= 0):
    sum = sum + number
    count = count + 1
    number = int(input(‘Gimme a number ’)
average = sum / count
```

Combining Iterations patterns: A slightly more complex example

- Find the max odd number is a list of positive integers
  - *Filter out odd numbers using enumerations*
  - *Keep track of the max number so far using an accumulator variable*



Max of odd numbers

```
max = None
number = int(input(‘Give me a number ‘))
while (number >= 0):
    if(number % 2 == 1):                        #filter odd numbers
        if (max == None or number > max):       #accumulate the max seen so far
            max = number
    number = int(input(‘Give me another number’))
if (max == None):
    print(‘You did not give me any odd numbers’)
else:
    print(f’The largest odd number was {max}’)
```

THIS IS ENOUGH PYTHON TO CODE ANY PROGRAM

AT LEAST IN THEORY!!!

In practice, however…

Why is Structuring Programs Important?

- What we have learnt so far is very powerful
  - *We can express any computation with what we have learnt so far*

but …

- This approach does not scale
  - *Difficult to code correctly*
  - *Difficult to debug*
  - *Difficult to extend*

Forces us to understand the whole program to make local changes

- To solve large problems, we need to
  - *Identify patterns that can help us solve multiple problems*
  - *Break (decompose) the problem into sub-problems*
  - *Find ways to re-use solutions for sub-problems across multiple problems*

Structuring Code: Twin Primes

- Given 2 numbers a and b
- We compute if a and b are twin primes as follows:
  - Compute isAPrime such as isAPrime will be True if a is a prime, False otherwise
  - Compute isBPrime such as isBPrime will be True if b is a prime , False otherwise
  - a and b are twin primes if |a – b| == 2 and isAPrime and isBPrime
- We may know how to compute if a number is prime, but looks like we need to make copies of the code

Computing Primes with a break

```
value = int(input('Give me a number '))

isPrime = True
for  factor in range(2, value // 2 + 1):
    if (value % factor == 0):
        isPrime = False
        break
if isPrime:
       print(f'{value} is a prime number')
```

1. We need two copies of the code in the red box
2. In the first copy, we rename value to a and isPrime to isAPrime
3. In the second copy, we rename value to b and isPrime to isBPrime
4. The final if and print are not needed

Bisection Search

- Find the square root of a number
  - *Solve by enumeration*
  - *Can we do this faster*
- Let’s play a game
  - *I will think of a number between 0 and 100*
  - *You guess a number*
  - *I will tell you if your guess is correct*
  - *If your guess is wrong, I will tell you if my number is smaller or larger*
  - *Objective: guess my number correctly with a minimum number of guesses*

Newton Raphson Method

```
number = 25
epsilon = 0.0001
guess = int(input("What is your first estimate of the answer"))
while(abs(guess ** 2 - number) > epsilon):
    guess = (guess + number / guess)/2
print(f'The answer is : {guess}')
```

Structuring Code:
Generic Bisection Method

- Inputs
  - *A function f*
  - *A range a – b such that a < b and f(x) takes on a zero value somewhere between a and b (ie f(a) and f(b) have opposite signs)*
  - *A measure of tolerance: epsilon*
  - *A guessing function g(a,b) returns a number c between a and b*
- Output: a value x between a and b such that |f(x) | <= epsilon
- Method
  - *Guess = g(a,b)*
  - *While |f(Guess) | > epsilon:*
    - If (sign(f(Guess) == sign (f(a)) then a = Guess
    - Else b = Guess   # sign(f(Guess) == sign(f(b))
    - Guess = g(a,b)

Some Issues with This Code

- Works only for defined values of `number` and `epsilon`
- Computes only square roots
  - *What if we need cube roots?*
- What if we need to compute square roots repeatedly?
  - *Multiple copies of code?*
  - *What if we make an error in the algorithm?*
- What do we do if we find a better method to compute square roots?
  - *Better could mean faster or more accurate*

We would like to generalize and reuse this code

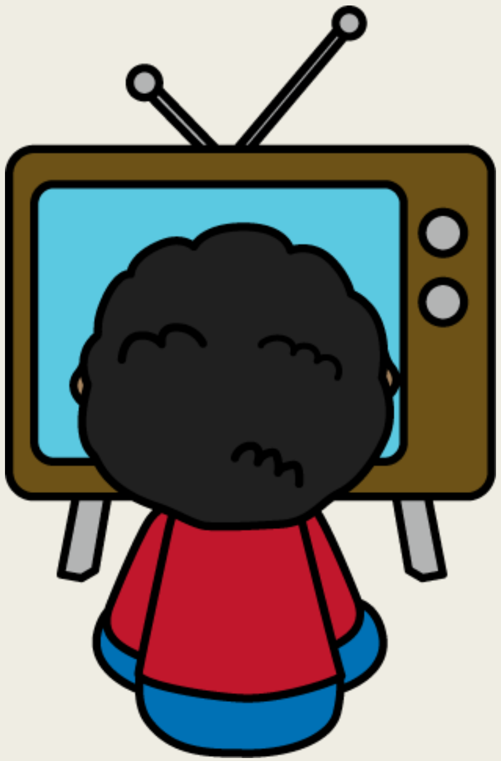
Writing Understandable Code

Divide and conquer

- Decompose the problem in one or more smaller problems
  - *At this point, we don’t need to understand how to solve each of these smaller problems*
  - *We only need to know how combine the solutions to these sub-problems into a solution for our problem*
- Solve the original problem by delegating work to these functions
  - *We only need to know how to only how each sub-problem behaves*
  - *Code each sub-problem into a function*

Abstraction

- Do you know how to use a Television?
- Do you know the details of how a Television works?



We can use something to achieve a task without understanding how it works

Decomposition

- Build a video wall with 9 TVs
- We need to ensure that the TVs are properly aligned and have thin (or no) edges
- We need to figure out how split a video image into 9 sub-images
- We need to ensure that the 9 TVs show the images at the same time

We are using multiple components to achieve a single task

Abstraction in Programming

- Hide away the details in your code into a black box when
  - *do not need to see the details*
  - *do not want to see the details*
  - *the details are tedious and don't contribute to overall understanding of your approach*
- Document intended behavior of your abstraction
  - *Detail intended inputs and expected outputs on a given input*
  - *This is called functional specification*

Decomposition In Programming

- Decompose (or break up) your code into understandable chunks that :
  - *are self contained*
  - *are intended to be re-usable*
  - *keep your code organized*
  - *keep your code coherent*

Functions: A Mechanism to Structure Code

- What is a Function
  - *"self contained" modules of code that accomplish a specific task.*
  - *usually "take in" data, process it, and "return" a result.*
  - *Once a function is written, it can be used over and over and over again.*
- What is a Function Call
  - *Functions can be "called" or ‘invoked” anywhere in the code where you need the specific task performed*
- We don't know (or care) **how** a function does what the specific task that it does
  - *We are only interested in knowing that it “does the task”!*

Functions are Tools

- Each tool is used to achieve a certain specific task
- You will need to use several different tools to complete your project
- You may not always understand how a tool works!

Libraries are toolboxes containing several different tools

Libraries of Functions

- Functions are reusable
- Several functions are available ready-made in libraries
- Popular libraries include
  - `numpy`
  - `pandas`
  - …
  - `tensorflow`
  - `pytorch`