Program Design with Loops

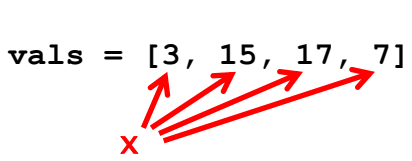
Computer Science 111
Boston University

Vahid Azadeh-Ranjbar, Ph.D.

Recall: Two Types of Loops

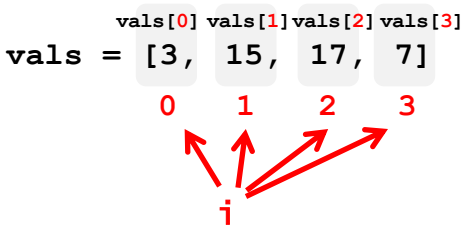
`for`	`while`
definite loop	***indefinite loop***
For a **known** number of repetitions	For an **unknown** number of repetitions

Recall: Two Types of `for` Loops




```
def sum(vals):
    result = 0
    for x in vals:
        result += x
    return result
```

element-based loop




```
def sum(vals):
    result = 0
    for i in range(len(vals)):
        result += vals[i]
    return result
```

index-based loop

Finding the Smallest Value in a List

- What if we needed to write a loop-based version of `min()`?

```
vals=[ 45, 80, 10, 30, 27, 50, 5, 15 ]
```

- What strategy should we use?
- What type of loop: `for` or `while`?
- What type of `for` loop: element-based or index-based?

Finding the Smallest Value in a List

- What if we needed to write a loop-based version of `min()`?

```
vals=[ 45, 80, 10, 30, 27, 50, 5, 15 ]
```

- What strategy should we use?
  iterate over the list, and keep track of the ***minimum so far***
- What type of loop: `for` or `while`?
  `for` loop
  a definite loop: we know how many repetitions we need
- What type of `for` loop: element-based or index-based?
  element-based:
  - we just need to process one element at a time
  - we don't need the indices

How should we fill in the blank to initialize m?

```
vals=[ 45, 80, 10, 30, 27, 50, 5, 15 ]
```

`m` is the "min so far"

```
def minval(vals):
    m = _____
    for x in vals:
        if x < m:
            m = x
    return m
```

A. `0`
B. `vals[0]`
C. either A or B will work
D. neither A nor B will work

How should we fill in the blank to initialize m?

```
vals=[ 45, 80, 10, 30, 27, 50, 5, 15 ]
```

m is the "min so far"

0 doesn't work, because it may be smaller than all of the values in the list.

```
def minval(vals):
    m = _________
    for x in vals:
        if x < m:
            m = x
    return m
```

A. 0

B. `vals[0]`

C. either A or B will work

D. neither A nor B will work

Finding the Smallest Value in a List

```
vals=[ 45, 80, 10, 30, 27, 50, 5, 15 ]
```

m is the "min so far"

m = 45

```
def minval(vals):
    m = vals[0]
    for x in vals:
        if x < m:
            m = x
    return m
```

Finding the Smallest Value in a List

```
vals=[ 45, 80, 10, 30, 27, 50, 5, 15 ]
```

m is the "min so far"


```
def minval(vals):
    m = vals[0]
    for x in vals:
        if x < m:
            m = x
    return m
```

Finding the Smallest Value in a List

```
vals=[ 45, 80, 10, 30, 27, 50, 5, 15 ]
```

m is the "min so far"


```
def minval(vals):
    m = vals[0]
    for x in vals:
        if x < m:
            m = x
    return m
```

Finding the Smallest Value in a List

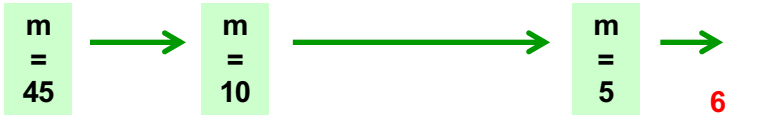
```
vals = [ 45, 80, 10, 30, 27, 50, 5, 15 ]
```

m is the "min so far"


```
def minval(vals):
    m = vals[0]
    for x in vals:
        if x < m:
            m = x
    return m
```

Finding the *Position* of the Smallest Value

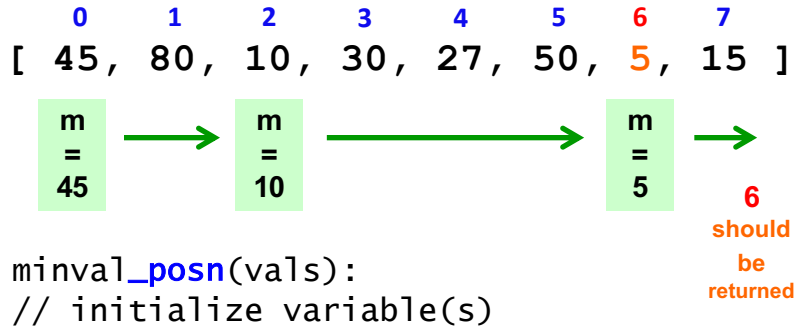
```
  0   1   2   3   4   5   6   7
[ 45, 80, 10, 30, 27, 50, 5, 15 ]
```




```
def minval_posn(vals):
    m = vals[0]
    for x in vals:
        if x < m:
            m = x
    return m
```

What if we want to know *where* the smallest value is found?

Finding the *Position* of the Smallest Value




```
def minval_posn(vals):
    // initialize variable(s)
    for i in range(len(vals)):
        if _________:
            // update var(s)
    return ________
```

What if we want to know *where* the smallest value is found?

we need an index-based loop!

Simpler

```
vals = [3, 15, 17, 7]
```

x

```
def sum(vals):
    result = 0
    for x in vals:
        result += x
    return result
```

element-based loop

More Flexible

vals[0] vals[1] vals[2] vals[3]

```
vals = [3, 15, 17, 7]
```

0 1 2 3

i

```
def sum(vals):
    result = 0
    for i in range(len(vals)):
        result += vals[i]
    return result
```

index-based loop

Determining if a Number is Prime

- Write a function `is_prime(n)` that:
  - returns `True` if `n` is prime
  - returns `False` otherwise
- Use a loop to check all possible divisors.
  - What are they?
  - For example, what divisors do we need to check for 41?

    `2, 3, 4, 5, 6, 7, 8, ..., 37, 38, 39, 40`

Determining if a Number is Prime

- Write a function `is_prime(n)` that:
  - returns `True` if `n` is prime
  - returns `False` otherwise
- Use a loop to check all possible divisors.
  - What are they?
  - For example, what divisors do we need to check for 41?

    `2, 3, 4, 5, 6, 7, 8, ..., 37, 38, 39, 40`

    `square root of 41 = 6.403124...`
- What type of loop should we use?
- What type of `for` loop?

Determining if a Number is Prime

- Write a function `is_prime(n)` that:
  - returns `True` if `n` is prime
  - returns `False` otherwise
- Use a loop to check all possible divisors.
  - What are they?
  - For example, what divisors do we need to check for 41?

    `2, 3, 4, 5, 6, 7, 8, ..., 37, 38, 39, 40`

    `square root of 41 = 6.403124...`
- What type of loop should we use?
  `for` loop – we know the sequence of values that we need to check
- What type of `for` loop?
  element-based – we don't need the indices

Determining if a Number is Prime

- Write a function `is_prime(n)` that:
  - returns `True` if `n` is prime
  - returns `False` otherwise

```
def is_prime(n):
    max_div = int(math.sqrt(n))    # max possible divisor

    # try all possible divisors
    for div in ____________________:
        if ________________:
            return ________   # when can we return "early"?

    # If we get here, what must be the case?
    return ____________
```

Determining if a Number is Prime

- Write a function `is_prime(n)` that:
  - returns `True` if `n` is prime
  - returns `False` otherwise

```
def is_prime(n):
    max_div = int(math.sqrt(n))    # max possible divisor

    # try all possible divisors
    for div in range(2, max_div + 1):
        if n % div == 0:
            return False      # return "early" if not prime

    # If we get here, n must be prime.
    return True
```

Does this version work?

- Write a function `is_prime(n)` that:
  - returns `True` if `n` is prime
  - returns `False` otherwise

```
def is_prime(n):
    max_div = int(math.sqrt(n))    # max possible divisor

    # try all possible divisors
    for div in range(2, max_div + 1):
        if n % div == 0:
            return False
        else:
            return True
```

Does this version work?

- Write a function `is_prime(n)` that:
  - returns `True` if `n` is prime
  - returns `False` otherwise

```
def is_prime(n):
    max_div = int(math.sqrt(n))    # max possible divisor

    # try all possible divisors
    for div in range(2, max_div + 1):
        if n % div == 0:
            return False
        else:
            return True    # no! returns too early!
```

Another Sample Problem

- `any_below(vals, cutoff)`
  - should return `True` if *any* of the values in `vals` is < `cutoff`
  - should return `False` otherwise
- examples:
  - `any_below([50, 18, 25, 30], 20)` should return `True`
  - `any_below([50, 18, 25, 30], 10)` should return `False`

Another Sample Problem

- `any_below(vals, cutoff)`
  - should return `True` if *any* of the values in `vals` is < `cutoff`
  - should return `False` otherwise
- examples:
  - `any_below([50, 18, 25, 30], 20)` should return `True`
  - `any_below([50, 18, 25, 30], 10)` should return `False`
- How should this method be implemented using a loop?

```
def any_below(vals, cutoff):
    for ____ in _____________:
        if _________________:
```

Which of these works?

A.

```
def any_below(vals, cutoff):
    for x in vals:
        if x >= cutoff:
            return False

    return True
```

B.

```
def any_below(vals, cutoff):
    for x in vals:
        if x < cutoff:
            return True

    return False
```

C.

```
def any_below(vals, cutoff):
    for x in vals:
        if x < cutoff:
            return True
        else:
            return False
```

D. more than one of them

Which of these works?

A.

```
def any_below(vals, cutoff):
    for x in vals:
        if x >= cutoff:
            return False

    return True
```

C.

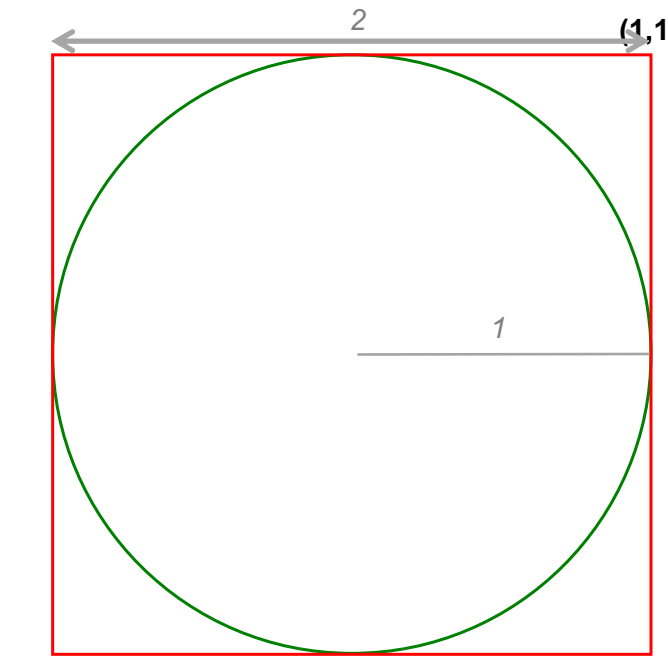
```
def any_below(vals, cutoff):
    for x in vals:
        if x < cutoff:
            return True
        else:
            return False
```

B.

```
def any_below(vals, cutoff):
    for x in vals:
        if x < cutoff:
            return True

    return False
```

D. more than one of them



Estimating π
by dart throwing!

$$\frac{\text{area of } \bigcirc}{\text{area of } \square} = \frac{\pi}{4}$$

$$\pi = \frac{4 * \text{area of } \bigcirc}{\text{area of } \square}$$

$$\pi \sim \frac{4 * \text{darts in } \bigcirc}{\text{darts in } \square}$$

"hits"

"throws"

Loops: `for` or `while`?

`pi_one(e)` — ***e == how close to π we need to get***

`pi_two(n)` — ***n == number of darts to throw***

Which function will use which kind of loop?

Loops: `for` or `while`?

`pi_one(e)` — ***e == how close to π we need to get***

use a `while` loop

`pi_two(n)` — ***n == number of darts to throw***

use a `for` loop

Which function will use which kind of loop?

Thinking in Loops

`for`	`while`
definite iteration	***indefinite iteration***
For a **known** number of repetitions	For an **unknown** number of repetitions