

Debugging skills

- Treat as a search problem: looking for explanation for incorrect behavior
 - Study available data – both correct test cases and incorrect ones
 - Form an hypothesis consistent with the data
 - Design and run a repeatable experiment with potential to refute the hypothesis
 - Keep record of experiments performed: use narrow range of hypotheses

Debugging as search

- Want to narrow down space of possible sources of error
- Design experiments that expose intermediate stages of computation (use print statements!), and use results to further narrow search
- Binary search can be a powerful tool for this

```
def isPal(x):  
    assert type(x) == list  
    temp = x  
    temp.reverse  
    if temp == x:  
        return True  
    else:  
        return False
```

```
def silly(n):  
    for i in range(n):  
        result = []  
        elem = raw_input('Enter element: ')  
        result.append(elem)  
    if isPal(result):  
        print('Yes')  
    else:  
        print('No')
```

Stepping through the tests

- Suppose we run this code:
 - We try the input 'abcba', which succeeds
 - We try the input 'palinnilap', which succeeds
 - But we try the input 'ab', which also 'succeeds'
- Let's use binary search to isolate bug(s)
- Pick a spot about halfway through code, and devise experiment
 - Pick a spot where easy to examine intermediate values

```
def isPal(x):  
    assert type(x) == list  
    temp = x  
    temp.reverse  
    if temp == x:  
        return True  
    else:  
        return False
```

```
def silly(n):  
    for i in range(n):  
        result = []  
        elem = raw_input('Enter element: ')  
        result.append(elem)  
    print(result)  
    if isPal(result):  
        print('Yes')  
    else:  
        print('No')
```



Stepping through the tests

- At this point in the code, we expect (for our test case of 'ab'), that result should be a list ['a', 'b']
- We run the code, and get ['b'].
- Because of binary search, we know that at least one bug must be present earlier in the code
- So we add a second print

```
def isPal(x):  
    assert type(x) == list  
    temp = x  
    temp.reverse  
    if temp == x:  
        return True  
    else:  
        return False
```

```
def silly(n):  
    for i in range(n):  
        result = []  
        elem = raw_input('Enter element: ')  
        result.append(elem)  
        print(result)  
    if isPal(result):  
        print('Yes')  
    else:  
        print('No')
```



Stepping through

- When we run with our example, the print statement returns
 - ['a']
 - ['b']
- This suggests that result is not keeping all elements
 - So let's move the initialization of result outside the loop and retry


```
def isPal(x):  
    assert type(x) == list  
    temp = x  
    temp.reverse  
    if temp == x:  
        return True  
    else:  
        return False
```

```
def silly(n):  
    result = []  
    for i in range(n):  
        elem = raw_input('Enter element: ')  
        result.append(elem)  
        print(result)  
    if isPal(result):  
        print('Yes')  
    else:  
        print('No')
```



Stepping through

- So this now shows we are getting the data structure result properly set up, but we still have a bug somewhere
 - A reminder that there may be more than one problem!
 - This suggests second bug must lie below print statement; let's look at isPal
 - Pick a point in middle of code, and add print statement again

```
def isPal(x):  
    assert type(x) == list  
    temp = x  
    temp.reverse  
    print(temp, x)  
    if temp == x:  
        return True  
    else:  
        return False
```

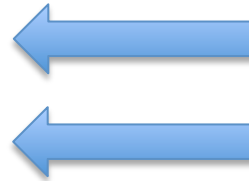


```
def silly(n):  
    result = []  
    for i in range(n):  
        elem = raw_input('Enter element: ')  
        result.append(elem)  
    if isPal(result):  
        print('Yes')  
    else:  
        print('No')
```

Stepping through

- At this point in the code, we expect (for our example of 'ab') that x should be ['a', 'b'], but temp should be ['b', 'a'], however they both have the value ['a', 'b']
- So let's add another print statement, earlier in the code

```
def isPal(x):  
    assert type(x) == list  
    temp = x  
    print(temp, x)  
    temp.reverse  
    print(temp, x)  
    if temp == x:  
        return True  
    else:  
        return False
```

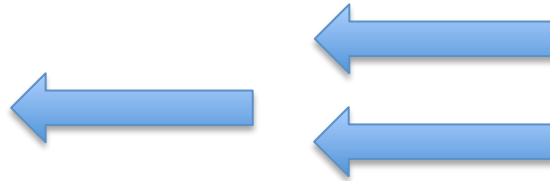


```
def silly(n):  
    result = []  
    for i in range(n):  
        elem = raw_input('Enter element: ')  
        result.append(elem)  
    if isPal(result):  
        print('Yes')  
    else:  
        print('No')
```

Stepping through

- And we see that `temp` has the same value before and after the call to `reverse`
- If we look at our code, we realize we have committed a standard bug – we forgot to actually invoke the `reverse` method
 - Need `temp.reverse()`
- So let's make that change and try again

```
def isPal(x):  
    assert type(x) == list  
    temp = x  
    print(temp, x)  
    temp.reverse()  
    print(temp, x)  
    if temp == x:  
        return True  
    else:  
        return False
```



```
def silly(n):  
    result = []  
    for i in range(n):  
        elem = raw_input('Enter element: ')  
        result.append(elem)  
    if isPal(result):  
        print('Yes')  
    else:  
        print('No')
```

Stepping through

- But now when we run on our simple example, both x and temp have been reversed!!
- We have also narrowed down this bug to a single line. The error must be in the reverse step
- In fact, we have an aliasing bug – reversing temp has also caused x to be reversed
 - Because they are referring to the same object


```
def isPal(x):
```

```
    assert type(x) == list
```

```
    temp = x[:]
```

```
    print(temp, x)
```

```
    temp.reverse()
```

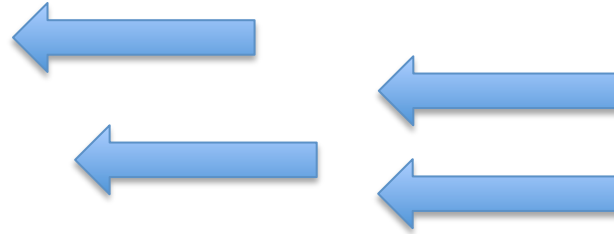
```
    print(temp, x)
```

```
    if temp == x:
```

```
        return True
```

```
    else:
```

```
        return False
```



```
def silly(n):
```

```
    result = []
```

```
    for i in range(n):
```

```
        elem = raw_input('Enter element: ')
```

```
        result.append(elem)
```

```
    if isPal(result):
```

```
        print('Yes')
```

```
    else:
```

```
        print('No')
```

Stepping through

- And now running this shows that before the reverse step, the two variables have the same form, but afterwards only temp is reversed.
- We can now go back and check that our other tests cases still work correctly

Some pragmatic hints

- Look for the usual suspects
- Ask why the code is doing what it is, not why it is not doing what you want
- The bug is probably not where you think it is – eliminate locations
- Explain the problem to someone else
- Don't believe the documentation
- Take a break and come back to the bug later