DECOMPOSITION, ABSTRACTION, FUNCTIONS

(download slides and .py files to follow along)

6.100L Lecture 7

Ana Bell

AN EXAMPLE: the SMARTPHONE

- A black box, and can be viewed in terms of
  - Its inputs
  - Its outputs
  - How outputs are related to inputs, without any knowledge of its internal workings
  - Implementation is “opaque” (or black)

AN EXAMPLE: the SMARTPHONE ABSTRACTION

- User **doesn't know the details** of how it works
  - We **don't need to know how something works** in order to know how to use it
- User **does know the interface**
  - Device converts a sequence of screen touches and sounds into expected useful functionality
- Know **relationship** between input and output

ABSTRACTION ENABLES DECOMPOSITION

- 100’s of distinct parts
- Designed and made by different companies
  - Do not communicate with each other, other than specifications for components
  - May use same subparts as others
- Each component maker has to know **how its component interfaces** to other components
- Each component maker can **solve sub-problems independent of other parts**, so long as they provide specified inputs
- True for hardware and for software

BIG IDEA

Apply abstraction (black box) and decomposition (split into self-contained parts) to programming!

SUPPRESS DETAILS with ABSTRACTION

- In programming, want to think of piece of code as **black box**
  - Hide tedious coding details from the user
  - Reuse black box at different parts in the code (no copy/pasting!)
- **Coder creates details**, and designs interface
- **User** does **not need or want** to see details

SUPPRESS DETAILS with ABSTRACTION

- Coder achieves abstraction with a **function (or procedure)**
- You've already been using functions!
- **A function** lets us capture code within a black box
  - Once we create function, it will produce an output from inputs, while hiding details of how it does the computation

```
max(1,4)
abs(-3)
len("mom's spaghetti")
```

SUPPRESS DETAILS with ABSTRACTION

- A function has **specifications**, captured using **docstrings**
- Think of a **docstring** as "contract" between coder and user:
  - If user provides **input** that satisfies stated conditions, function will produce **output** according to specs, including indicated **side effects**
  - Not typically enforced in Python (we'll see assertions later), but user relies on coder's work satisfying the contract

```
abs(-3)
```

```
abs(
abs(x, /)

Return the absolute value of the argument.
```

CREATE STRUCTURE with DECOMPOSITION

- Given the idea of black box abstraction, use it to **divide code into modules** that are:
  - **Self-contained**
  - Intended to be **reusable**
- Modules are used to:
  - **Break up** code into logical pieces
  - Keep code **organized**
  - Keep code **coherent** (readable and understandable)
- In this lecture, achieve decomposition with **functions**
- In a few lectures, achieve decomposition with **classes**
- Decomposition relies on abstraction to enable construction of complex modules from simpler ones

FUNCTIONS

- Reusable pieces of code, called **functions** or **procedures**
- Capture steps of a computation so that we can use with any input
- A function is just some **code written in a special, reusable way**

FUNCTIONS

- **Defining a function** tells Python some code now exists in memory
- Functions are only useful when they are **run** (“**called**” or “**invoked**”)
- You write a function once but can run it many times!
- Compare to code in a file
  - It doesn’t run when you load the file
  - It runs when you hit the run button

FUNCTION CHARACTERISTICS

- Has a **name**
  - (think: variable bound to a function object)
- Has (formal) **parameters** (0 or more)
  - The inputs
- Has a **docstring** (optional but recommended)
  - A comment delineated by `"""` (triple quotes) that provides a **specification** for the function – contract relating output to input
- Has a **body**, a set of instructions to execute when function is called
- **Returns** something
  - Keyword `return`

HOW to WRITE a FUNCTION

keyword

name

parameters or arguments

specification, docstring

```
def is_even( i ):
    """
    Input: i, a positive int
    Returns True if i is even, otherwise False
    """
    if i%2 == 0:
        return True
    else:
        return False
```

indentation defines extent of function body

body

return keyword tells function to give back a value

HOW TO THINK ABOUT WRITING A FUNCTION

- **What** is the problem?
  - Given an int, call it `i`, want to know if it is even
  - Use this to write the function name and specs

```
def is_even( i ):
    """
    Input: i, a positive int
    Returns True if i is even, otherwise False
    """
```

HOW TO THINK ABOUT WRITING A FUNCTION

- How to **solve** the problem?
  - Can check that remainder when divided by 2 is 0
  - Think about what value you need to give back

```
def is_even( i ):
    """
    Input: i, a positive int
    Returns True if i is even, otherwise False
    """
    if i%2 == 0:
        return True
    else:
        return False
```

HOW TO THINK ABOUT WRITING A FUNCTION

- Can you make the code **cleaner**?
  - i%2 is a Boolean that evaluates to True/False already

```
def is_even( i ):
    """
    Input: i, a positive int
    Returns True if i is even, otherwise False
    """
    return i%2 == 0
```

BIG IDEA

At this point, all we’ve done is make a function object

HOW TO CALL (INVOKE) A FUNCTION

Name of the function

Values for parameters of the function

```
is_even(3)
is_even(8)
```

- That's all!

HOW TO CALL (INVOKE) A FUNCTION

Replaced by the return!

```
is_even(3)
is_even(8)
```

- That's all!

ALL TOGETHER IN A FILE

- This code might be in one file

Function definition

```
def is_even( i ):
    return i%2 == 0
```

Function call

```
is_even(3)
```

WHAT HAPPENS when you CALL a FUNCTION?

- Python replaces:
  formal parameters in function def with **values from function call**
  i replaced with **3**

i mapped to 3

```
def is_even( i ):
    return i%2 == 0
```

```
is_even(3)
```

WHAT HAPPENS when you CALL a FUNCTION?

- Python replaces:
  formal parameters in function def with **values from function call**
  i replaced with **3**
- Python **executes expressions in the body** of the function
  - `return 3%2 == 0`

```
def is_even( i ):
    return i%2 == 0
```

keyword

expression to evaluate and return to invoker

`3%2 == 0` is `False`

```
is_even(3)
```

Replaced by False

WHAT HAPPENS when you CALL a FUNCTION?

- Python replaces:
  formal parameters in function def with **values from function call**
  i replaced with **3**

```
def is_even( i ):
    return i%2 == 0
```

```
is_even(3)
print(is_even(3))
```

Replaced by False

BIG IDEA

A function's code only runs when you call (aka invoke) the function

YOU TRY IT!

- Write code that satisfies the following specs

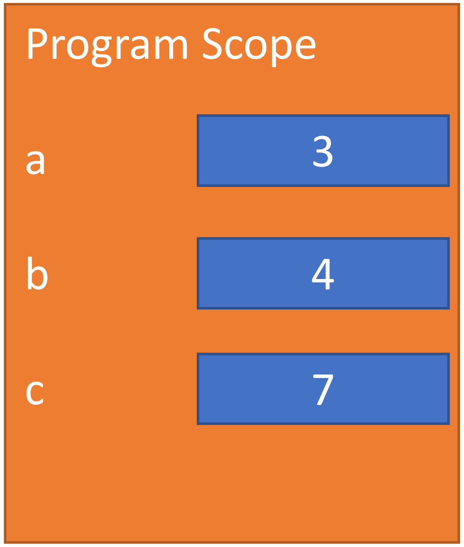
```
def div_by(n, d):
    """ n and d are ints > 0
        Returns True if d divides n evenly and False otherwise """
```

Test your code with:

- n = 10 and d = 3
- n = 195 and d = 13

ZOOMING OUT
(no functions)

```
a = 3
b = 4
c = a+b
```

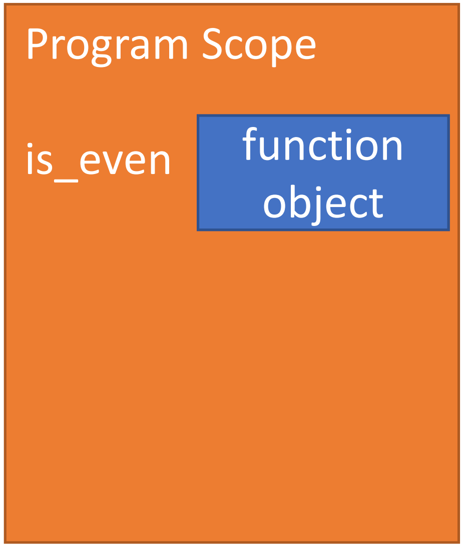


ZOOMING OUT

This is my "black box"

```
def is_even( i ):
    print("inside is_even")
    return i%2 == 0
```

```
a = is_even(3)
b = is_even(10)
c = is_even(123456)
```



This is me telling my black box to do something

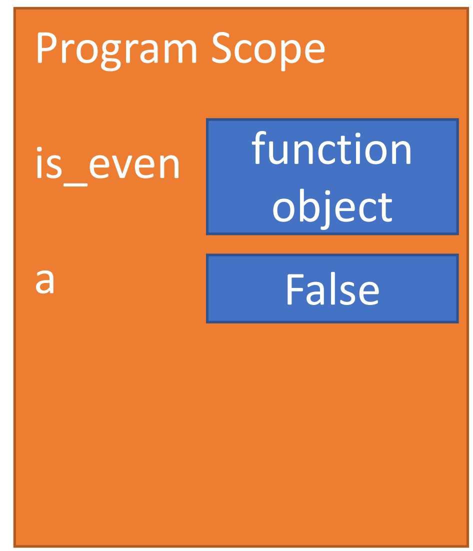
ZOOMING OUT

This is my “black box”

```
def is_even( i ):
    print("inside is_even")
    return i%2 == 0
```

```
a = is_even(3)
b = is_even(10)
c = is_even(123456)
```

One function call



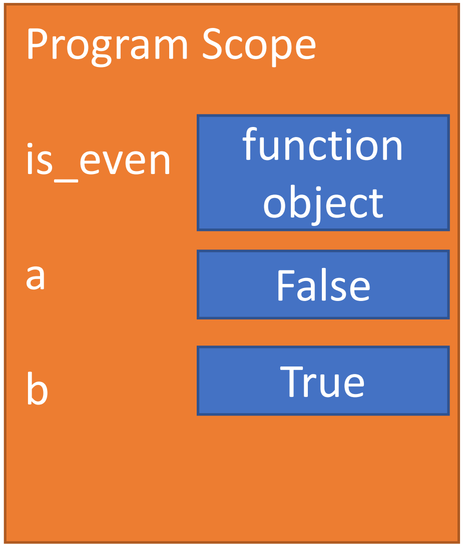
ZOOMING OUT

This is my “black box”

```
def is_even( i ):
    print("inside is_even")
    return i%2 == 0
```

```
a = is_even(3)
b = is_even(10)
c = is_even(123456)
```

One function call



ZOOMING OUT

This is my “black box”

```
def is_even( i ):
    print("inside is_even")
    return i%2 == 0
```

```
a = is_even(3)
b = is_even(10)
c = is_even(123456)
```

One function call

Program Scope	
is_even	function object
a	False
b	True
c	True

INSERTING FUNCTIONS IN CODE

- Remember how expressions are replaced with the value?
- The **function call** is **replaced** with the **return value**!

```
print("Numbers between 1 and 10: even or odd")

for i in range(1,10):
    if is_even(i):
        print(i, "even")
    else:
        print(i, "odd")
```

ANOTHER EXAMPLE

- Suppose we want to add all the odd integers between (and including) `a` and `b`

- What is the **input**?
  - Values for a and b
- What is the **output**?
  - The sum_of_odds

```
def sum_odd(a, b):
    # your code here
    return sum_of_odds
```

BIG IDEA

Don’t write code right away!

PAPER FIRST

- Suppose we want to add all the odd integers between (and including) `a` and `b`

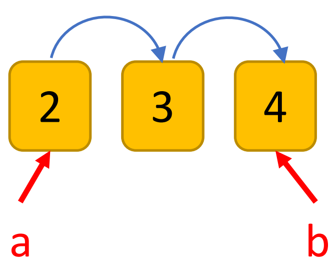
- Start with a **simple example on paper**
- Systematically solve the example

```
def sum_odd(a, b):
    # your code here
    return sum_of_odds
```

SIMPLE TEST CASE

- Suppose we want to add all the odd integers between (and including) `a` and `b`

- Start with a simple example on paper
- a = 2 and b = 4
    - sum_of_odds should be 3



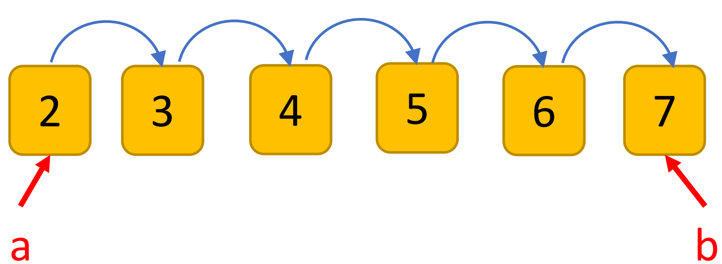

```
def sum_odd(a, b):
    # your code here
    return sum_of_odds
```

MORE COMPLEX TEST CASE

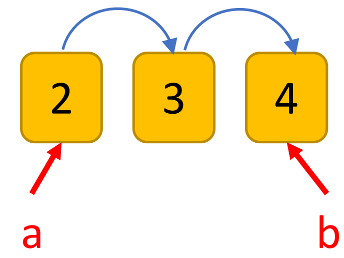
- Suppose we want to add all the odd integers between (and including) `a` and `b`

```
def sum_odd(a, b):
    # your code here
    return sum_of_odds
```

- Start with a simple example on paper
- a = 2 and b = 7
  - sum_of_odds should be 15



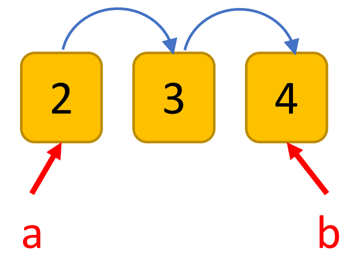
SOLVE SIMILAR PROBLEM



- Start by looking at each number between (and including) a and b
- A similar problem that is easier that you know how to do?
  - Add **ALL** numbers between (and including) a and b
  - Start with this

```
def sum_odd(a, b):
    # your code here
    return sum_of_odds
```

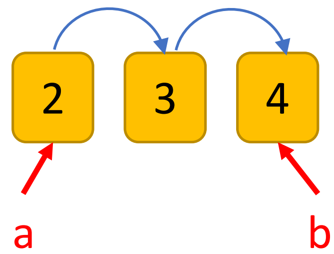
CHOOSE BIG-PICTURE STRUCTURE



- Add **ALL** numbers between (and including) a and b
  - It's a loop
- while or for?
  - Your choice

```
def sum_odd(a, b):
    # your code here
    return sum_of_odds
```

WRITE the LOOP (for adding all numbers)



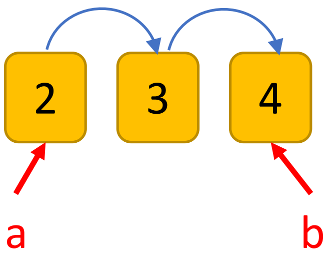
`for` LOOP

```
def sum_odd(a, b):
    for i in range(a, b):
        # do something
    return sum_of_odds
```

`while` LOOP

```
def sum_odd(a, b):
    i = a
    while i <= b:
        # do something
        i += 1
    return sum_of_odds
```

DO the SUMMING
(for adding all numbers)



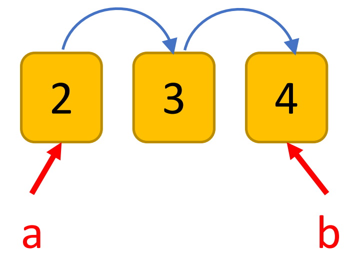
`for` LOOP

```
def sum_odd(a, b):
    for i in range(a, b):
        sum_of_odds += i
    return sum_of_odds
```

`while` LOOP

```
def sum_odd(a, b):
    i = a
    while i <= b:
        sum_of_odds += i
        i += 1
    return sum_of_odds
```

INITIALIZE the SUM (for adding all numbers)



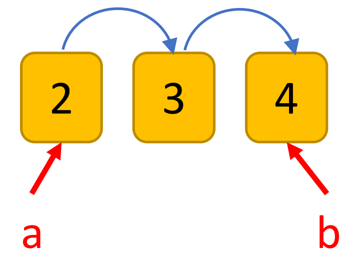
`for` LOOP

```
def sum_odd(a, b):
    sum_of_odds = 0
    for i in range(a, b):
        sum_of_odds += i
    return sum_of_odds
```

`while` LOOP

```
def sum_odd(a, b):
    sum_of_odds = 0
    i = a
    while i <= b:
        sum_of_odds += i
        i += 1
    return sum_of_odds
```

TEST!
(for adding all numbers)



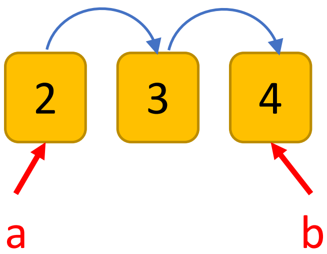
`for` LOOP

```
def sum_odd(a, b):
    sum_of_odds = 0
    for i in range(a, b):
        sum_of_odds += i
    return sum_of_odds


print(sum_odd(2,4))
```

`while` LOOP

```
def sum_odd(a, b):
    sum_of_odds = 0
    i = a
    while i <= b:
        sum_of_odds += i
        i += 1
    return sum_of_odds


print(sum_odd(2,4))
```

WEIRD RESULTS…
(for adding all numbers)



`for` LOOP

```
def sum_odd(a, b):
    sum_of_odds = 0
    for i in range(a, b):
        sum_of_odds += i
    return sum_of_odds



print(sum_odd(2,4))
```

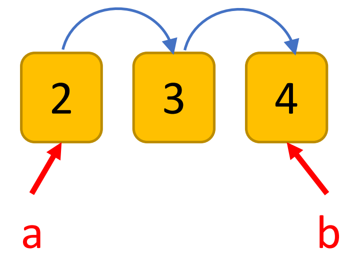
5

`while` LOOP

```
def sum_odd(a, b):
    sum_of_odds = 0
    i = a
    while i <= b:
        sum_of_odds += i
        i += 1
    return sum_of_odds


print(sum_odd(2,4))
```

9

DEBUG! aka ADD PRINT STATEMENTS (for adding all numbers)



`for` LOOP

```
def sum_odd(a, b):
    sum_of_odds = 0
    for i in range(a, b):
        sum_of_odds += i
        print(i, sum_of_odds)
    return sum_of_odds
```

```
2 2
3 5
```

```
print(sum_odd(2,4))
```

5

`while` LOOP

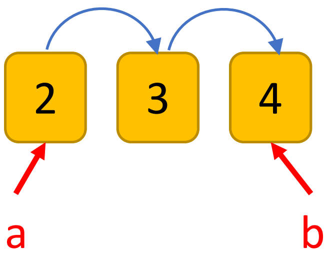
```
def sum_odd(a, b):
    sum_of_odds = 0
    i = a
    while i <= b:
        sum_of_odds += i
        print(i, sum_of_odds)
        i += 1
    return sum_of_odds
```

```
2 2
3 5
4 9
```

```
print(sum_odd(2,4))
```

9

FIX for LOOP END INDEX (for adding all numbers)



`for` LOOP

```
def sum_odd(a, b):
    sum_of_odds = 0
    for i in range(a, b+1):
        sum_of_odds += i
        print(i, sum_of_odds)
    return sum_of_odds


print(sum_odd(2,4))
```

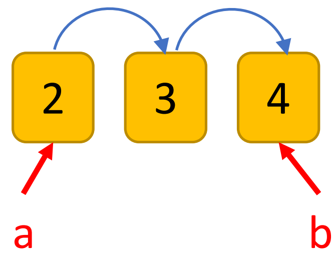
9

`while` LOOP

```
def sum_odd(a, b):
    sum_of_odds = 0
    i = a
    while i <= b:
        sum_of_odds += i
        print(i, sum_of_odds)
        i += 1
    return sum_of_odds


print(sum_odd(2,4))
```

9

ADD IN THE ODD PART!



`for` LOOP

```
def sum_odd(a, b):
    sum_of_odds = 0
    for i in range(a, b+1):
        if i%2 == 1:
            sum_of_odds += i
            print(i, sum_of_odds)
    return sum_of_odds

print(sum_odd(2,4))
```

3

`while` LOOP

```
def sum_odd(a, b):
    sum_of_odds = 0
    i = a
    while i <= b:
        if i%2 == 1:
            sum_of_odds += i
            print(i, sum_of_odds)
        i += 1
    return sum_of_odds
print(sum_odd(2,4))
```

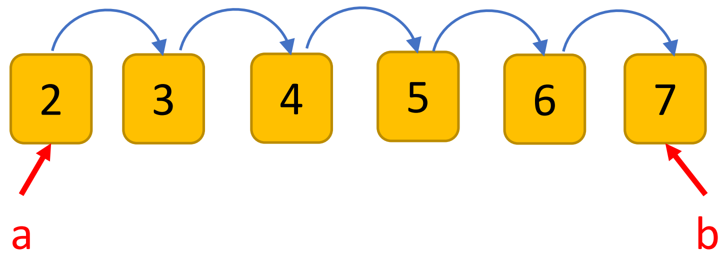
3

BIG IDEA

Solve a simpler problem first.

Add functionality to the code later.

TRY IT ON ANOTHER EXAMPLE



`for` LOOP

```
def sum_odd(a, b):
    sum_of_odds = 0
    for i in range(a, b+1):
        if i%2 == 1:
            sum_of_odds += i
    return sum_of_odds


print(sum_odd(2,7))
```

15

`while` LOOP

```
def sum_odd(a, b):
    sum_of_odds = 0
    i = a
    while i <= b:
        if i%2 == 1:
            sum_of_odds += i
        i += 1
    return sum_of_odds

print(sum_odd(2,7))
```

15

PYTHON TUTOR

- Also a great debugging tool

BIG IDEA

Test code often.
Use prints to debug.

YOU TRY IT!

- Write code that satisfies the following specs

```
def is_palindrome(s):
    """ s is a string
    Returns True if s is a palindrome and False otherwise
    """
```

For example:

- If s = "222" returns True
- If s = "2222" returns True
- If s = "abc" returns False

SUMMARY

- Functions allow us to **suppress detail** from a user
- Functions **capture computation** within a black box
- A programmer writes functions with
  - 0 or more **inputs**
  - Something to **return**
- A function only **runs when it is called**
- The entire function call is **replaced with the return value**
  - Think expressions! And how you replace an entire expression with the value it evaluates to.