Wrap-up Session

ASPC-A, Session 9

Syllabus

- Creating and running Python programs
- Variables, data types
- Input and output
- Conditionals
- Lists
- Loops
- Iteration
- Recursion

Operations on numbers

`-a`	**Negate** a number
`a + b`	**Add** two numbers
`a - b`	**Subtract** a number from a number
`a * b`	**Multiply** two numbers
`a / b`	**Divide** a number by another number
`a // b`	**Quotient** of a and b, **rounded down**.
`a % b`	**Remainder** when a is divided by b (also called a **modulo** b, also written as $a \bmod b$)
`a ** b`	Calculate a^b.

Operations on strings

`s1 + s2`	**Concatenate** (or **combine**) two strings
`s[i]`	Get a string of length one containing the character at index i.
`s[i:j]`	Get a string with the characters from indices i to $j - 1$.
`s[i:]`	Get a string with the characters from index i until the end.
`s[:j]`	Get a string with the characters from the start to index $j - 1$.

Type conversion functions

`int(a)`	Converts a to an `int`
`float(a)`	Converts a to a `float`
`str(a)`	Converts a to a `string`

Examples:

```
print(str(25 + 23) + "orange")
print(int("1000") + 2 )
print(float("54.2") * 2)
```

Comparison Operators

- Compare both numbers and strings and get a `boolean`:

`a == b`	`a` **equals** `b`
`a != b`	`a` is **not equal to** `b`

- Compare numbers and get a `boolean`:

`a < b`	`a` is **less than** `b`
`a > b`	`a` is **greater than** `b`
`a <= b`	`a` is **less than or equal to** `b`
`a >= b`	`a` is **greater than or equal to** `b`

Logical Operators

- Operations on Boolean values which give Booleans:

Syntax	Name	What it does
`not a`	Logical NOT	Negates a `boolean` value Turns `True` to `False` and turns `False` to `True`.
`a and b`	Logical AND	If both `a` and `b` are `True`, yields `True` else, yields `False`
`a or b`	Logical OR	If any of `a` or `b` is `True`, yields `True` else, yields `False`

The `if` Statement

```
if boolean :
    # code that you want to run
    # in the event that the boolean
    # above is true
    # this block of text is indented
# anything not indented afterwards is
# after the if statement
```

if-else

```
if boolean1:
    # do stuff here if boolean1
    # above is true
else:
    # do stuff here if boolean1
    # above is false
# code not indented is outside the if-else
```

elif

We can write the code as:

```
if boolean1:
    # boolean1 is true
elif boolean2:
    # boolean1 is false
    # boolean2 is true
```

We can read "elif" as "else if".

Functions

- **Syntax to declare a function**

```
def name( arguments ):
    # insert code here
```

Using lists

- Like any other data, you can **assign** a list to a variable.

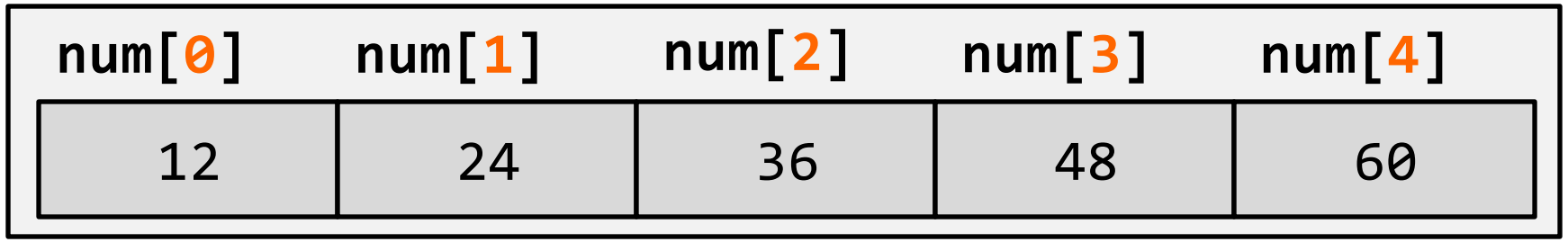
```
myList = [1, 2, 3, 4, 5, 6]
pets = ["cat","dog"]
answer = []
```

Accessing items in lists

`L[i]`	Gets the element at index i of the list L.
`L[i:j]`	Get the portion of the list L from indices i to $j-1$.
`L[i:]`	Get the portion of the list L from index i until the end.
`L[:j]`	Get the portion of the list L from the start to index $j-1$.

- Lists in Python use **zero-based indexing**.
- The first element has an index of zero.

```
num = [12, 24, 36, 48, 60]
```



Other operations on a list

`L.append(x)`	Adds x as a new item at the end of L.
`L.clear()`	Removes all elements from L.
`len(L)`	Gets the number of items in L.

```
myList = [12, 3, 4, 5, 0]
print(len(myList))
myList.append(7)
print(myList)
myList.clear()
print(myList)
```

Output:

```
5
[12, 3, 4, 5, 0, 7]
[]
```

`range()`

- The **range** function in Python generates a sequence of numbers using **very little code.**
  - `range(a)` generates $0,1,2,\ldots,a-1$.
  - `range(8)` generates `0, 1, 2, 3, 4, 5, 6, 7`
  - `range(10)` generates `0, 1, 2, 3, 4, 5, 6, 7, 8, 9`
  - `range(1000)` generates `0, 1, ..., 998, 999`
- Ranges are not the same as lists, and cannot be printed like lists.
  - Try running `print(range(5))`

Patterns of Iteration

- We will be discussing many general problems concerning iteration and lists
  - Not going to cover *every problem ever*, but enough to give you a good **toolkit**
- The patterns:

Range for-loop	Any
Map	All
Filter	Reduce
Counter	

Range For-loop & Map

Range for loop: do something *num_times* times.

```
for counter in range(num_times):
    do_something
```

Map: do an operation on all elements.

```
result = []
for item in sequence:
    result_item = apply_operation(item)
    result.append(result_item)
```

Filter & Counter

Filter: only keep elements which satisfy a condition

```
result = []
for item in sequence:
    if condition_on_item:
        result.append(item)
```

Counter: count number of items in a list which satisfy a condition:

```
count = 0
for item in sequence:
  if condition_on_item:
    count = count + 1 #increase value of count by 1
```

The Any Pattern

```
ans = False
for item in input_sequence:
    if condition_on_item:
        ans = True
```

- Replace red parts with relevant code

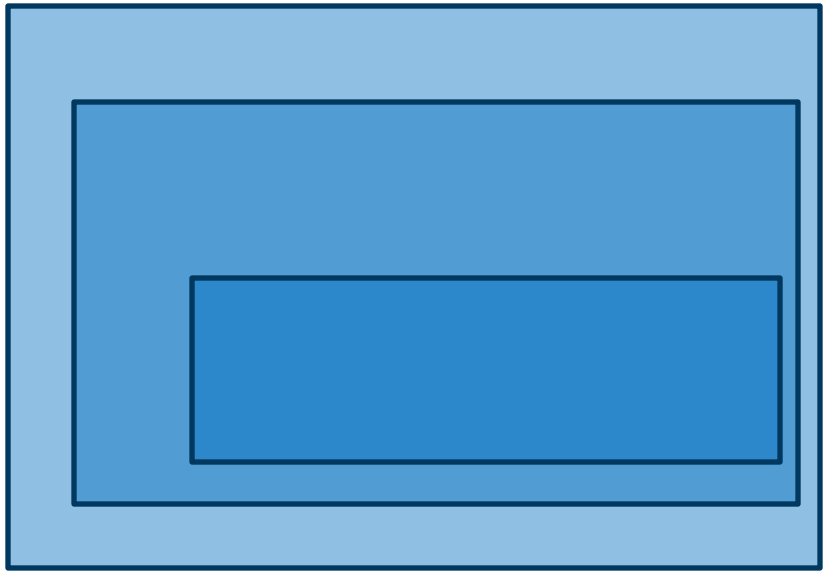
The All Pattern

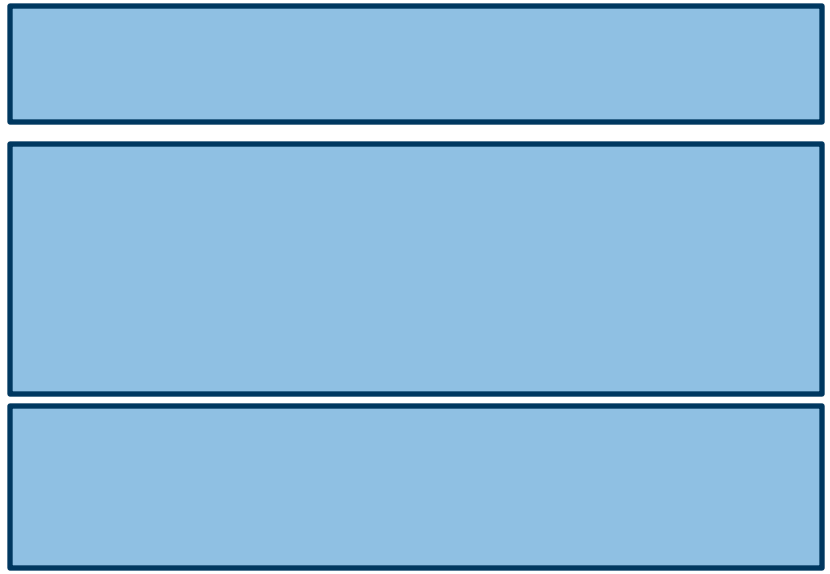
```
ans = True
for item in input_sequence:
    if not condition_holds_for(item):
        ans = False
```

- Replace red parts with relevant code

Combining Patterns

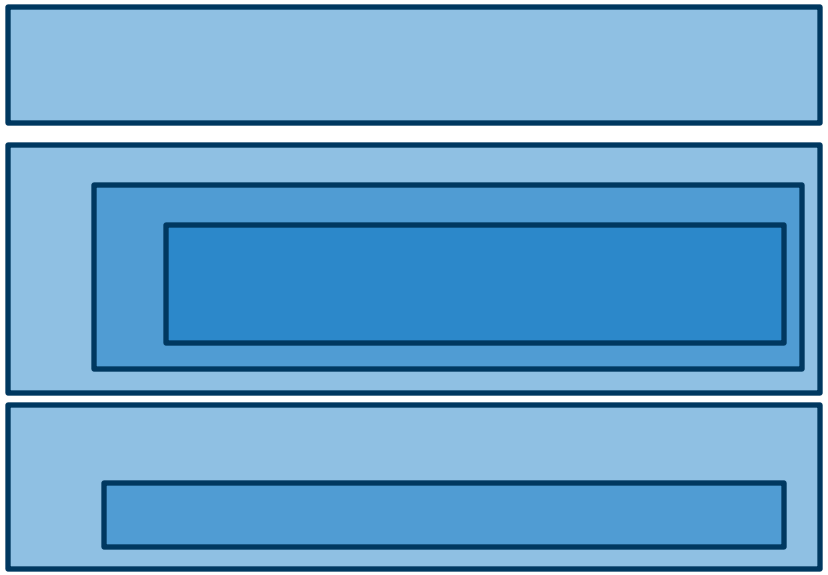
- There are two general ways to combine patterns: **nesting** patterns inside other patterns, and **chaining** multiple patterns one after another.





Combining Patterns

- These two approaches are often used together.



Library

- **Collection of functions and variables** which do related things.
- Programs can use libraries to perform common tasks.
- Python has many **built-in functions** to support mathematical operations.
- To use some functions, you have to **import** the required library first
  - `import math`

abs()

```
abs( number )
```

- Returns the absolute value of a number, defined as

$$|x| = \begin{cases} x \text{ if } x \geq 0 \\ -x \text{ if } x < 0 \end{cases}$$

```
someNum = -123.45
print( abs( someNum ) )
```

min() and max()

`min( A, B )`

- Returns the smallest value in the argument
- The argument can be a sequence or a list.

```
min( 3, 9 )

min( 3, 9, 5, 2, 5 )

min( [3, 9, 5, 2, 5] )
```

`max( A, B )`

- Returns the largest value in the argument.
- The argument can be a sequence or a list.

```
max( 3, 9 )

max( 3, 9, 5, 2, 5 )

max( [3, 9, 5, 2, 5] )
```

`math.pi`

- Python has a built-in constant for the value of pi. (π)

For any circle, $\pi = \frac{\text{circumference}}{\text{diameter}}$.

```
print( math.pi )
```

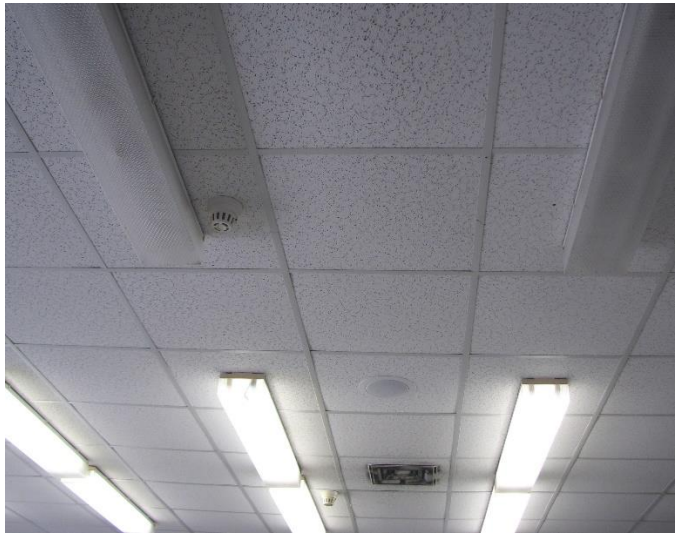
Output:

```
3.141592653589793
```

`math.floor()` and `math.ceil()`

- `math.floor( x )`
- Returns the largest integer number less than or equal to x

- `math.ceil( x )`
- Returns the smallest integer number greater than or equal to x



math.sqrt()

```
math.sqrt( x )
```

- Returns an approximation close to the square root of x.

```
print( math.sqrt( 2 ) )
print( math.sqrt( 26 ) )
```

Python's built-in sorting functions

`list.sort()`

- Sorts the given list.

`sorted(list)`

- Returns a sorted copy of the list, but does not modify the original list.
- Sorting functions will sort the list in **ascending order**.

Brute Force

- Use **counters/iteration patterns** to solve problems by **brute-forcing** through all possibilities.
- In other words, **try all possibilities**.
- Computers can do a lot of work using just loops!

Brute Force

Benefits

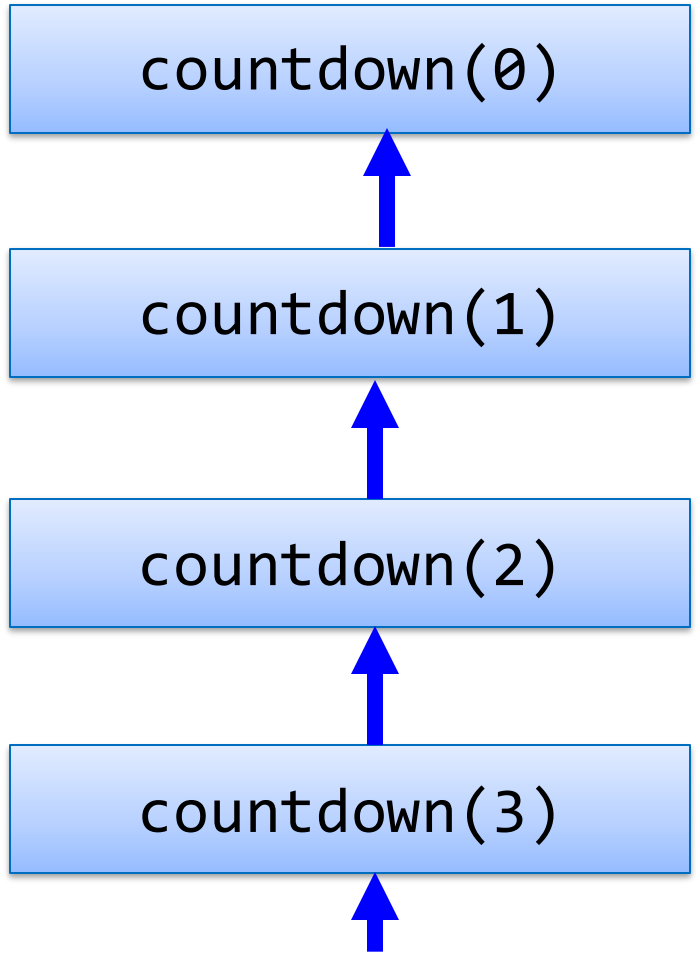
- No need to think about complicated math!
- The computer does (almost) all the work for you.

Problems

- It can be *slow*.
- Problem: Given n $(1 \leq n \leq 10^{12})$, output the sum of all integers from 1 to n.

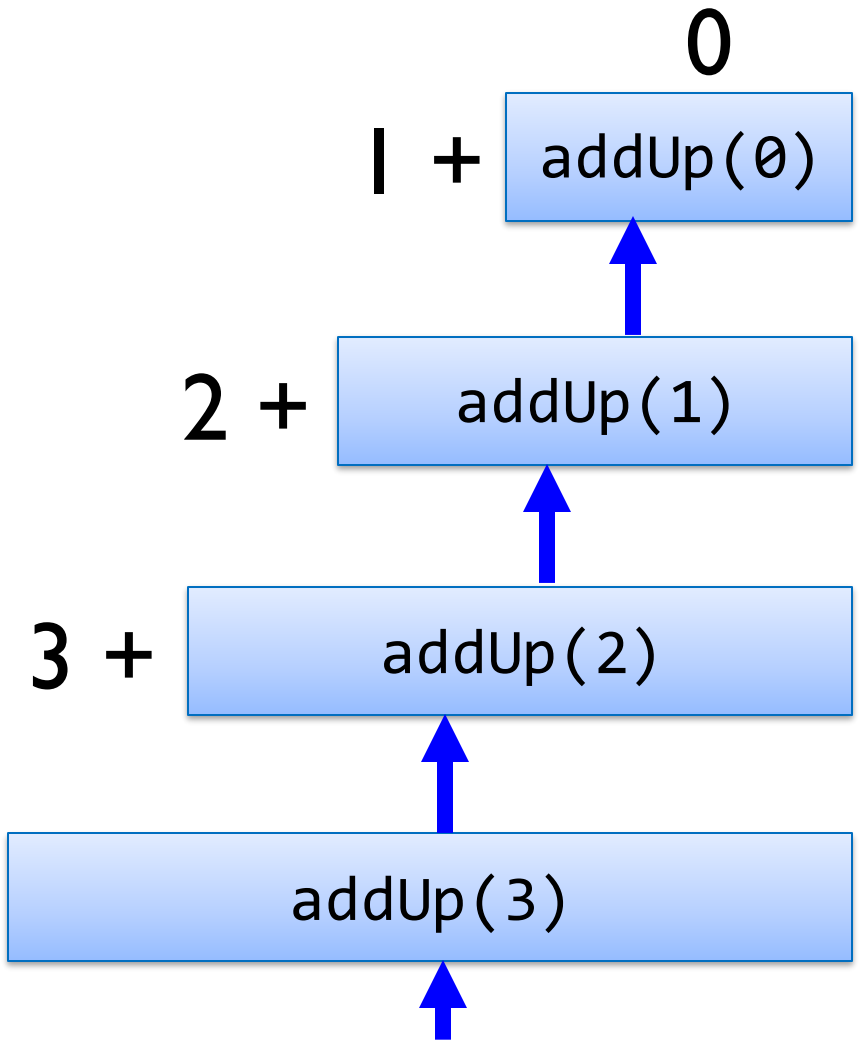
Recursion: Calling and Returning

- Calling `countdown()` will call countdown with a smaller argument.
- Once the argument is small enough, nothing new is called and the entire thing returns.



Recursion

- In each function call, we make the argument smaller while adding a number.
- We stop calling functions when the argument is small enough.
- We return the total of all the numbers that we add.
- Solve a problem by splitting the problem, then solving the individual parts.

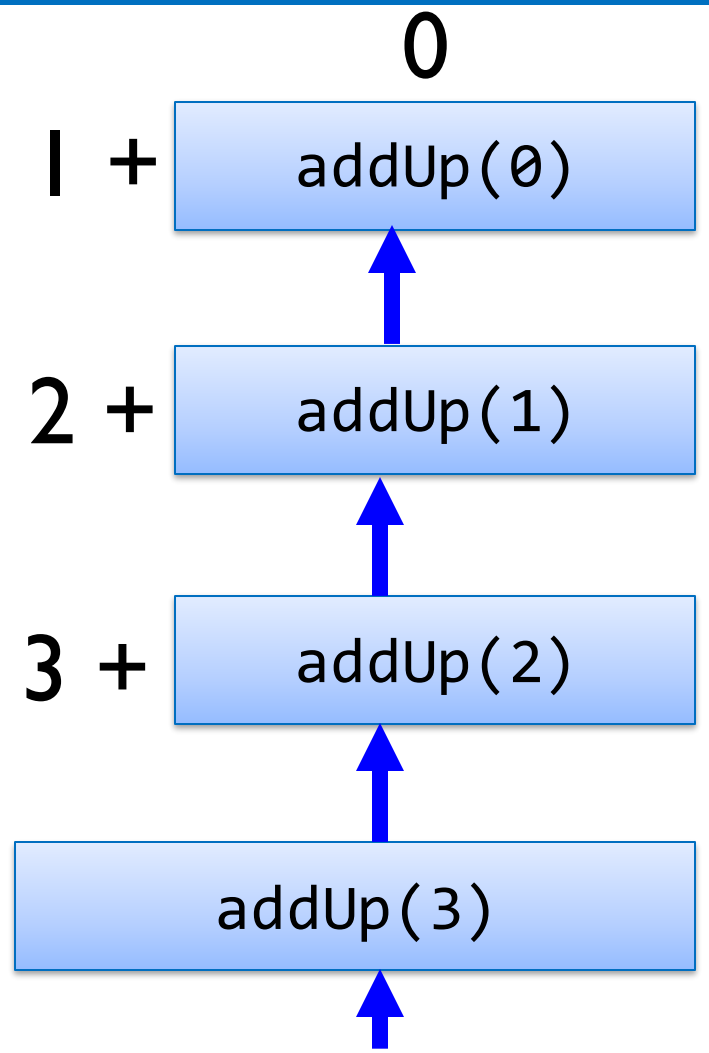


Recursion

```
def addUp( i ):
    if i == 0:
        return 0
    else:
        return i + addUp( i - 1 )
```

Base case: **smallest version of the problem**, we return the solution

Recursive case: we break down our problem into smaller cases, and **call the same function, with a smaller problem**



Syllabus

- Creating and running Python programs
  - Variables, data types
  - Input and output
- Conditionals and Functions
- Lists and Loops
- Design Patterns
- Libraries
- Brute Force
- Recursion

Thanks!


accenture
PLDT
stratpoint
FAST FORWARD TO THE FUTURE
MERALCO
Thinking Machines
Data Science