Questions of the day

- What other ways does Python have to manipulate strings, lists, and other data?
- How does recursion work?

String processing and recursion

William Hendrix

Lecture 3

Outline

- Review
  - Loops
  - File I/O
  - Lists and dictionaries
- String processing
- Recursion

Review: loops

- `for i in range(start, end, increment):`
  - `start` and `increment` are both optional
- `for i in my_list:`
  - Iterates through list elements
- `for key in my_dict:`
  - Iterates through dictionary keys
- `while` condition:
  - Syntax similar to `if`
  - Beware infinite loops!
- `break`: terminate loop immediately
- `continue`: skip to next iteration
- Loop `else`
  - Skipped on `break`

Review: file I/O

- `open(filename, mode='r')`
  - Need relative or absolute path if not same directory
- Read with:

  `file.read()`: everything

  `file.readlines()`: list of strings

  `for line in file:`: iterate through lines

  `file.readline()`: one line
  - All have `'\n'` at end of line
  - `line.rstrip()`: remove trailing whitespace
- Write: `'w'` mode

  `file.write(line) # with '\n'!`
  - Can use f-string
- `file.close()`
- `with open(...) as file:`

  `  # processing code`
  - Closes automatically

Review: data structures

- String: sequence of characters
  - 1-3 ' or "
  - `f'{...}'`: put expression and formatting in `{}`
    - `f'${money:.2f}'`
    - `f'{right_text:>8}`
- List: sequence of anything

  `[elem1, elem2, ...]`
- Dictionary: set of key-value pairs

  `{key1 : value1, key2 : value2, ...}`
- `len(coll)`: length/size of any collection
- `my_list[index]` or `my_dict[key]`
  - Negative indexes
  - String is immutable (~~`my_str[0] = 'X'`~~)
- `list.append(x), list/dict.pop(i)`

Slicing

- Allows selecting several characters at once
  - Strings or lists
- **Syntax:** `str[start:end]`
  - First character is `str[start]`
  - Goes up to, but does not include, index `end`
- **Examples:**

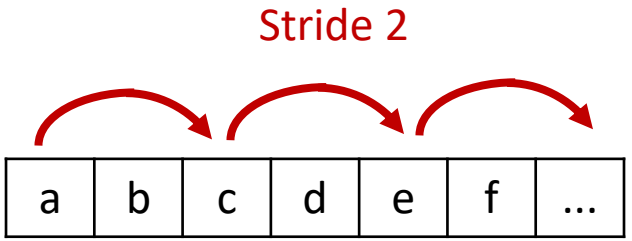
```
my_str = 'abcdef'
print(my_str[0:1]) # 'a'
print(my_str[1:2]) # 'b'
print(my_str[1:4]) # 'bcd'
print(my_str[2:6]) # 'cdef'
print(my_str[0:len(my_str)]) # 'abcdef'
# 0:len(str) is always entire string
```

Advanced slicing

- Can omit `start` and/or `end`
  - `str[start:]` or `str[:end]`
  - Starts at beginning, ends at end if omitted
- **Examples:**

```
my_str = 'abcdef'
print(my_str[:2]) # 'ab' (first two)
print(my_str[2:]) # 'cdef' (after first 2)
print(my_str[:])  # 'abcdef' (everything)
```

- Can also add a *stride* with extra `:`
- **Examples:**




```
my_str = 'abcdef'
print(my_str[0:len(my_str):2]) # 'ace'
print(my_str[1::2] # 'bdf'
print(my_str[::-1]) # 'fedcba'
```

Searching

- `str.index('x')`: index of `x` in `str`
  - Also works for lists
  - Returns first instance if x appears multiple times
  - Can find substrings:

```
i = str.index('the')
# str[i] == 't', str[i+1] == 'h', str[i+2] == 'e'
```

  - ...but not sublists:
    - `[1, 2, 3].index([2, 3]) # Error:  not present`
    - `[1, [2, 3]].index([2, 3]) # 1`

- `'x' in str`: whether `str` contains `x`
  - Behavior similar to `.index`
  - Also works for dictionaries
    - Returns whether LHS is a key

String processing example

- I'll write a script that prompts the user to type a word, then converts that word into *Pig Latin*

  Pig Latin rules

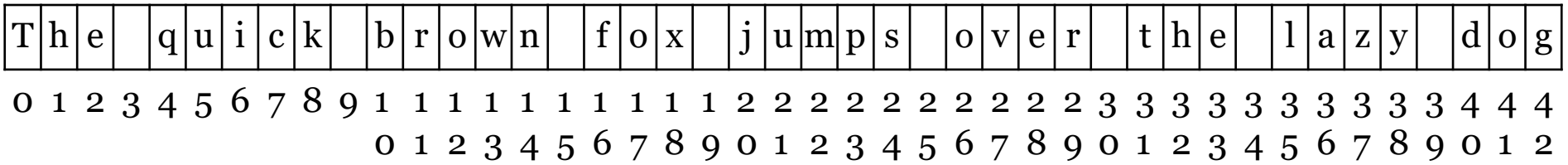
  – Move all consonants at the beginning of the word (up to first vowel) to the end of the word

  – Append *ay* as a suffix

- **Examples:**
  – can: *ancay*
  – i: *iay*
  – speak: *eakspay*
- *Hint:* write a function to translate a word to Pig Latin. Find the index of the first vowel (a, e, i, o, u) in the input

String processing exercise

- Write a script that extracts various strings from the sentence "The quick brown fox jumps over the lazy dog"
  - `'the lazy dog'`
  - `'abc'`
  - `'revo'`



Type checking

- `type(x)`
  - Returns data type of variable or expression

```
type(0) # int
type(""Hey!"") # str
type([1, 3, 5, 7, 9]) # list
type(my_var) # depends!
```

- `isinstance(x, type)`
  - Returns whether variable or expression has given type

```
isinstance(0, int) # True
isinstance(0, float) # False
isinstance(0., int) # False
```

split()

- `str.split('\t')`: converts string into list
  - First list entry is start of `str` up to first `\t`
  - Second entry is after first `\t` character to second `\t`, etc.
  - `str.split()`: default separator is space (`' '`)
- Some useful applications
  - `str.split()`: words in `str`
  - `str.split('\n')`: lines in str
  - `str.split('\t')`: columns in a tab-separated table
  - `str.split(',')`: comma-separated values
- **Example:**

```
words = 'it was the best of times, \
it was the worst of times'.split()
str = ''
for word in words[2, 6]: # the best of times,
    str = str + word
```

`join()`

- Inverse operation to `split()`
- `sep.join(list)`: joins strings in a list together
  - Concatenates all strings in `list` together
  - Separated by `sep`
- `''.join(list)`: concatenate
- `' '.join(list)`: combine with spaces (words)
- `'\n'.join(list)`: combine with new line (lines)

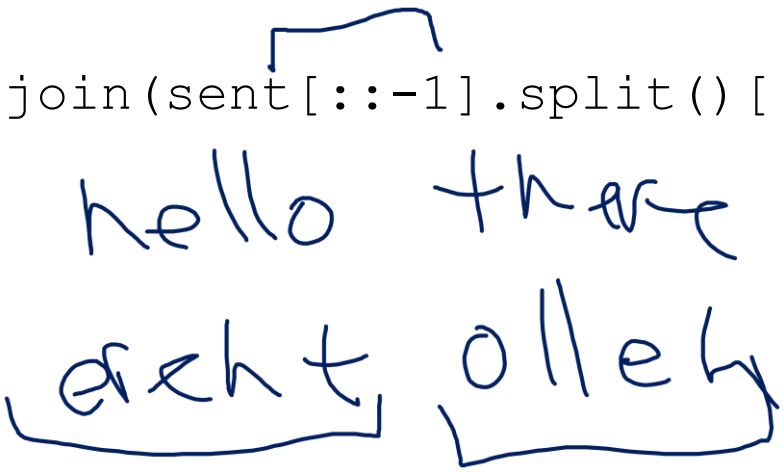
`split()/join()` examples

- Reversing the words in a sentence

```
rev = ' '.join(sent.split()[::-1])
```

- Reversing the letters in each word of a sentence

```
rev = ''
for word in sent.split():
    rev += word[::-1] + ' '
# or:
rev = ' '.join(sent[::-1].split()[::-1])
```



Split/join exercise

- Expand the Pig Latin translator so that it can translate an entire sentence
  - Assume no punctuation or capitalization

Tuples and sets

- Both support many collection operations (`len`, `for`, etc.)
- **tuple:** immutable list
- **Syntax:** `(elem1, elem2, ...)`
  - `()` optional
- Can be used to perform multiple assignment

```
x, y = y, x
```

- More efficient than lists
- No `[]` on LHS of assignment, no `.append` or `.pop`, etc.
- **set:** *unordered* collection of *unique* elements
- **Syntax:** `{elem1, elem2, ...}`
  - No `[]`; mainly used with `in`
  - `.add` instead of `.append`
- Faster with `in` than lists or tuples

List comprehensions

- Powerful technique to define a list
- **Syntax:** `my_list = [expr for var in iterable]`
  - Equivalent to:

```
my_list = []
for var in iterable:
    my_list.append(expr)
```

  - `expr` usually involves loop variable (`var`)
  - `iterable` can be `range`, `list`, file, etc.
- **Example:** changing a range into a string of numbers

```
', '.join([str(i) for i in range(1, 10)])
```

  - *Output:* `'1, 2, 3, 4, 5, 6, 7, 8, 9'`
- Also works for sets, dicts, and tuples

```
(str(i) for i in range(10))
{str(i) for i in range(10))
{i : str(i) for i in range(10)}
```

Namespaces

- Namespace: mapping of variable names to values
- Python usually has 4 namespaces (scope levels)
  - `local`: variables in a function
  - `enclosing`: used with functions defined inside functions
  - `global`: variables or functions not in a function
  - `builtin`: things Python loads automatically
    - E.g., `print()`, `input()`, `float()`, `str()`, `list()`
- For each variable: Python checks local scope first, then enclosing, then global, then builtins (LEGB)
- Python (and most languages) allow you do define variables with the same names in different scopes

```
str(42) # from built-in scope                    Built-in str
str = 'My string' # global scope                 Global str
def spam(str): # local scope                     Local str
    print(str)
spam(str)
```

Function call review

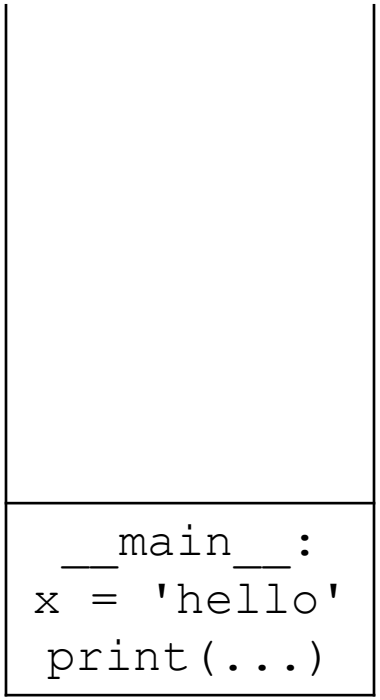
- When function is called:
  - Current function pauses
  - Python creates new `local` namespace
  - Start running function
- On return:
  - `local` variables erased
  - Return to calling function
  - Replace function call with return value
  - Resume function

```
def ham(a, b):
  return a * b

def spam():
  x = 6
  y = x + 1
  return ham(x, y)

x = "hello"
print(x)
print(spam())
```

Functions (simplified):



Function call review

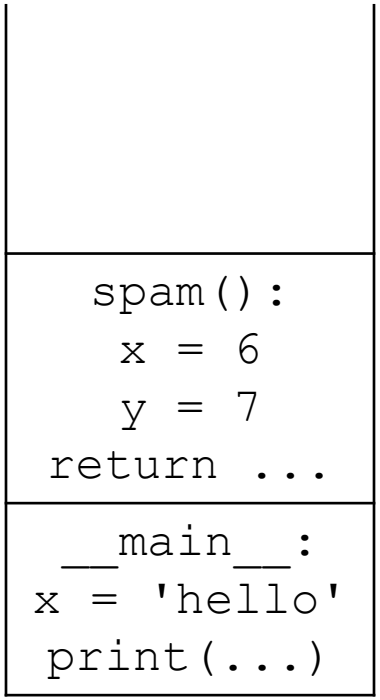
- When function is called:
  - Current function pauses
  - Python creates new `local` namespace
  - Start running function
- On return:
  - `local` variables erased
  - Return to calling function
  - Replace function call with return value
  - Resume function

```
def ham(a, b):
  return a * b

def spam():
  x = 6
  y = x + 1
  return ham(x, y)

x = "hello"
print(x)
print(spam())
```

Functions (simplified):



Function call review

- When function is called:
  - Current function pauses
  - Python creates new `local` namespace
  - Start running function
- On return:
  - `local` variables erased
  - Return to calling function
  - Replace function call with return value
  - Resume function

```
def ham(a, b):
  return a * b

def spam():
  x = 6
  y = x + 1
  return ham(x, y)

x = "hello"
print(x)
print(spam())
```

Functions (simplified):

```
ham():
 a = 6
 b = 7
return a*b
```

```
spam():
 x = 6
 y = 7
return ...
```

```
__main__:
x = 'hello'
print(...)
```

Function call review

- When function is called:
  - Current function pauses
  - Python creates new `local` namespace
  - Start running function
- On return:
  - `local` variables erased
  - Return to calling function
  - Replace function call with return value
  - Resume function

```
def ham(a, b):
  return a * b

def spam():
  x = 6
  y = x + 1
  return ham(x, y)

x = "hello"
print(x)
print(spam())
```

Functions (simplified):

```
ham():
 a = 6
 b = 7
return 6*7
```

```
spam():
 x = 6
 y = 7
return ...
```

```
__main__:
x = 'hello'
print(...)
```

Function call review

- When function is called:
  - Current function pauses
  - Python creates new `local` namespace
  - Start running function
- On return:
  - `local` variables erased
  - Return to calling function
  - Replace function call with return value
  - Resume function

```
def ham(a, b):
  return a * b

def spam():
  x = 6
  y = x + 1
  return ham(x, y)

x = "hello"
print(x)
print(spam())
```

Functions (simplified):

```
ham():
 a = 6
 b = 7
return 42
```

```
spam():
 x = 6
 y = 7
return ...
```

```
__main__:
x = 'hello'
print(...)
```

Function call review

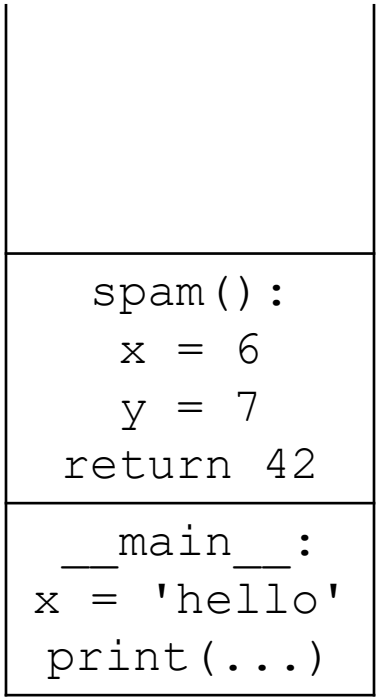
- When function is called:
  - Current function pauses
  - Python creates new `local` namespace
  - Start running function
- On return:
  - `local` variables erased
  - Return to calling function
  - Replace function call with return value
  - Resume function

```
def ham(a, b):
  return a * b

def spam():
  x = 6
  y = x + 1
  return ham(x, y)

x = "hello"
print(x)
print(spam())
```

Functions (simplified):



Function call review

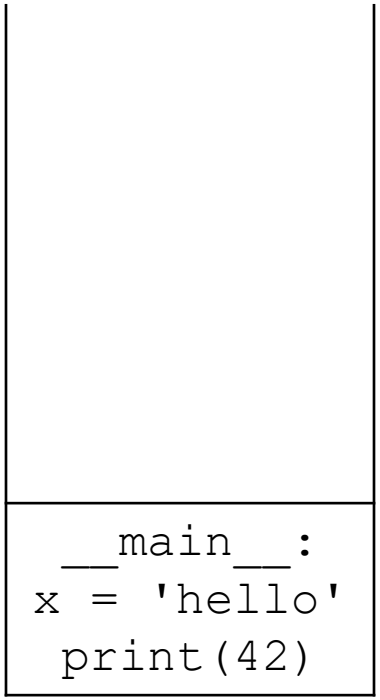
- When function is called:
  - Current function pauses
  - Python creates new `local` namespace
  - Start running function
- On return:
  - `local` variables erased
  - Return to calling function
  - Replace function call with return value
  - Resume function

```
def ham(a, b):
  return a * b

def spam():
  x = 6
  y = x + 1
  return ham(x, y)

x = "hello"
print(x)
print(spam())
```

Functions (simplified):



Recursive function

- Function that calls itself
- **Example:** calculating factorial
  - Product of 1 up to *n*
  - Formal definition: $n! = \begin{cases} 1, & \text{if } n = 0, \\ n(n-1)!, & \text{otherwise} \end{cases}$
- **Factorial example:** 4!

4! = 4 * 3!
= 4 * (3 * 2!)
= 4 * 3 * (2 * 1!)
= 4 * 3 * 2 * (1 * 0!)
= 4 * 3 * 2 * 1 * 1
= 24

Recursive function code

- Function that calls itself
- **Example:** calculating factorial
  - Product of 1 up to *n*
  - Formal definition: $$n! = \begin{cases} 1, & \text{if } n = 0, \\ n(n-1)!, & \text{otherwise} \end{cases}$$
- We can turn the formal definition into Python code

```
# Function to calculate n!
def fact(n):
  if n == 0:
    return 1 # 0! = 1
  else:
    return n * fact(n-1) # n! = n(n-1)!
```

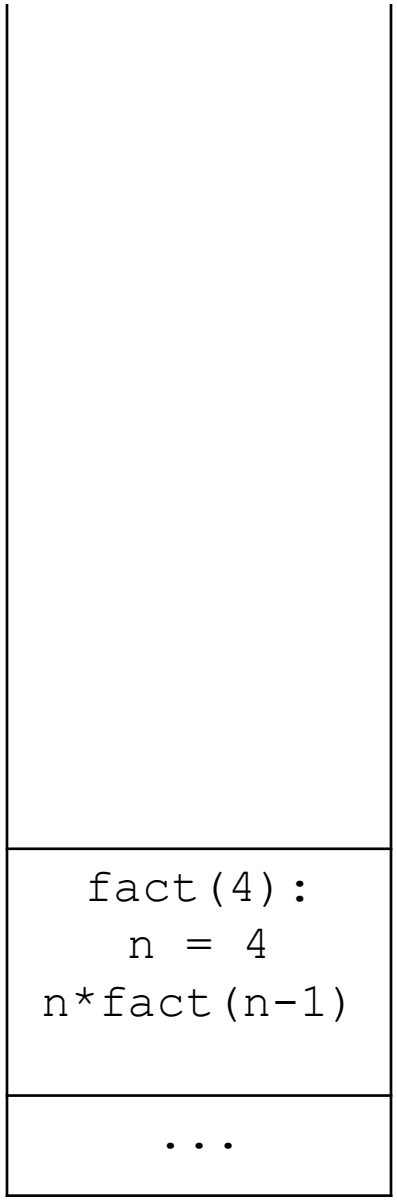
  - Calling `fact` inside `fact`: recursive!

Recursive trace example

- Function that calls itself
- **Example:**

```
def fact(n):
  if n == 0:
    return 1
  else:
    return n*fact(n-1)
```

- Tracing `fact(4)`:

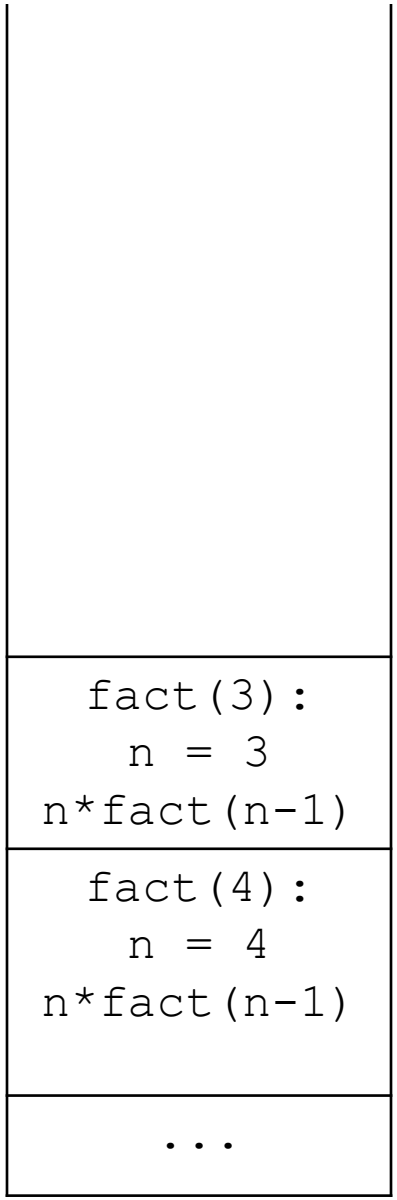


Recursive trace example

- Function that calls itself
- **Example:**

```
def fact(n):
  if n == 0:
    return 1
  else:
    return n*fact(n-1)
```

- Tracing `fact(4)`:

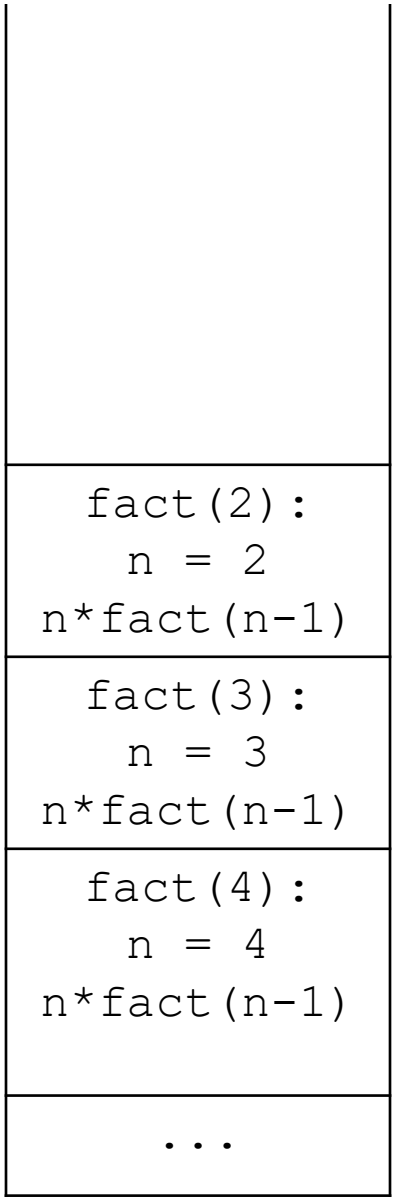


Recursive trace example

- Function that calls itself
- **Example:**

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n*fact(n-1)
```

- Tracing `fact(4)`:

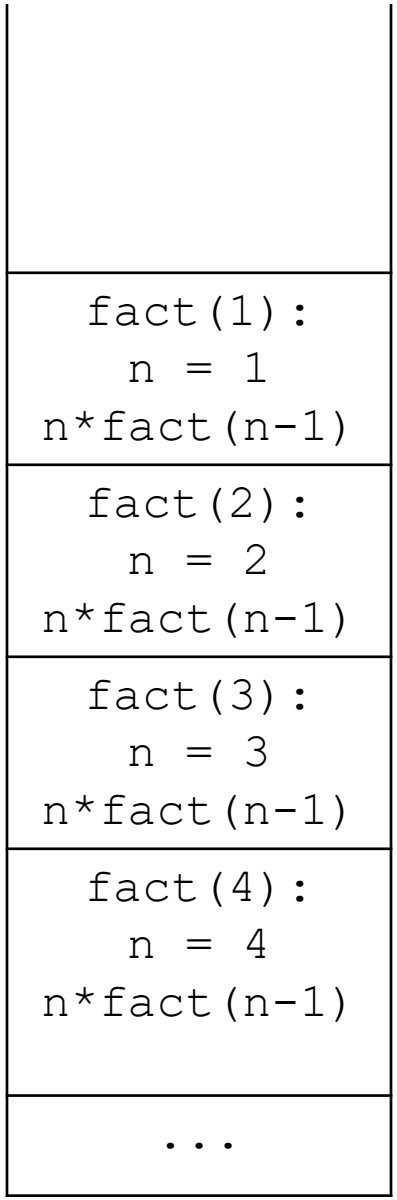


Recursive trace example

- Function that calls itself
- **Example:**

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n*fact(n-1)
```

- Tracing `fact(4)`:

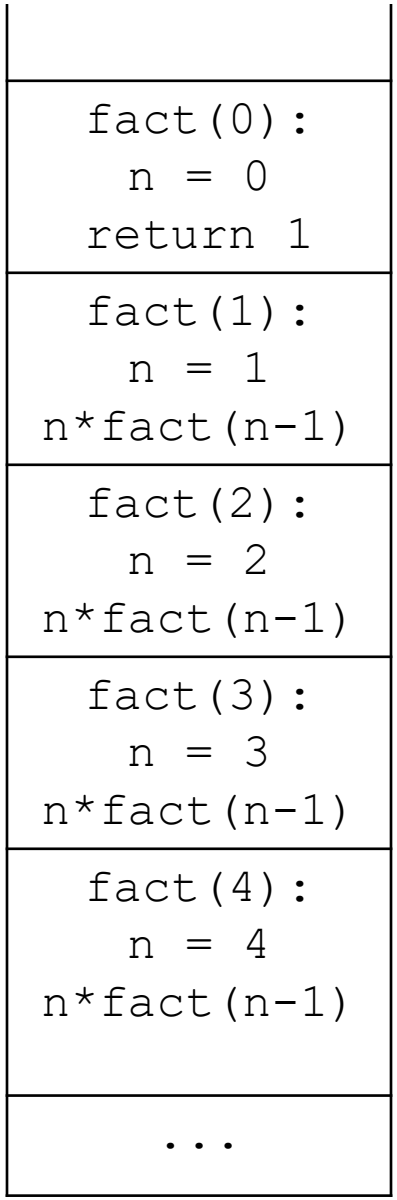


Recursive trace example

- Function that calls itself
- **Example:**

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n*fact(n-1)
```

- Tracing `fact(4)`:

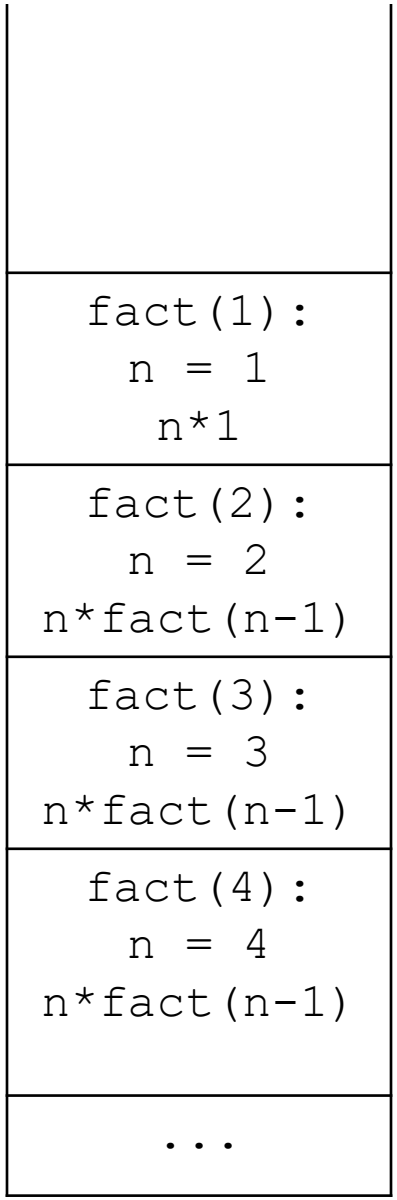


Recursive trace example

- Function that calls itself
- **Example:**

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n*fact(n-1)
```

- Tracing `fact(4)`:

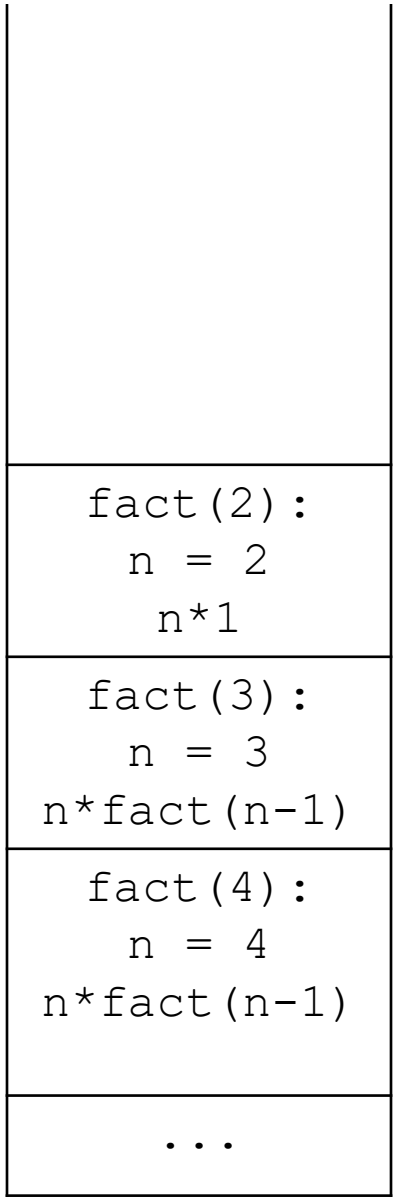


Recursive trace example

- Function that calls itself
- **Example:**

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n*fact(n-1)
```

- Tracing `fact(4)`:

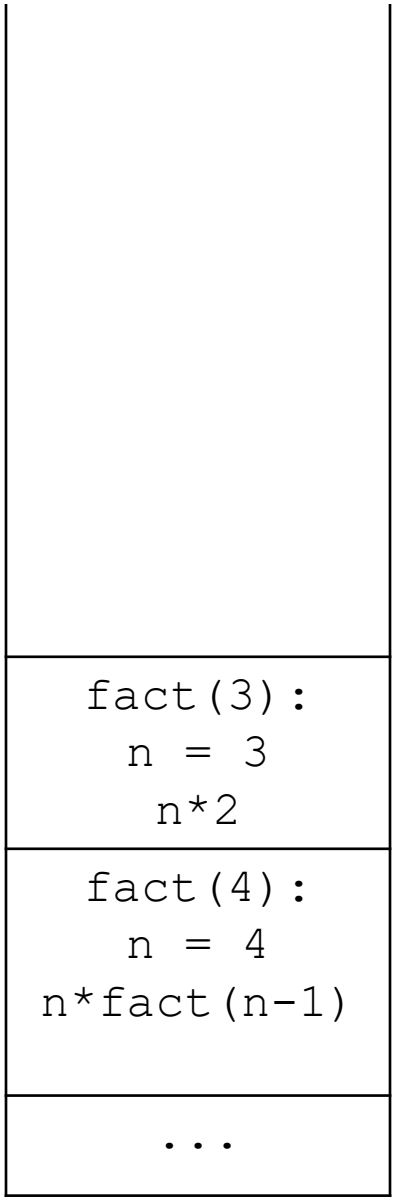


Recursive trace example

- Function that calls itself
- **Example:**

```
def fact(n):
  if n == 0:
    return 1
  else:
    return n*fact(n-1)
```

- Tracing `fact(4)`:

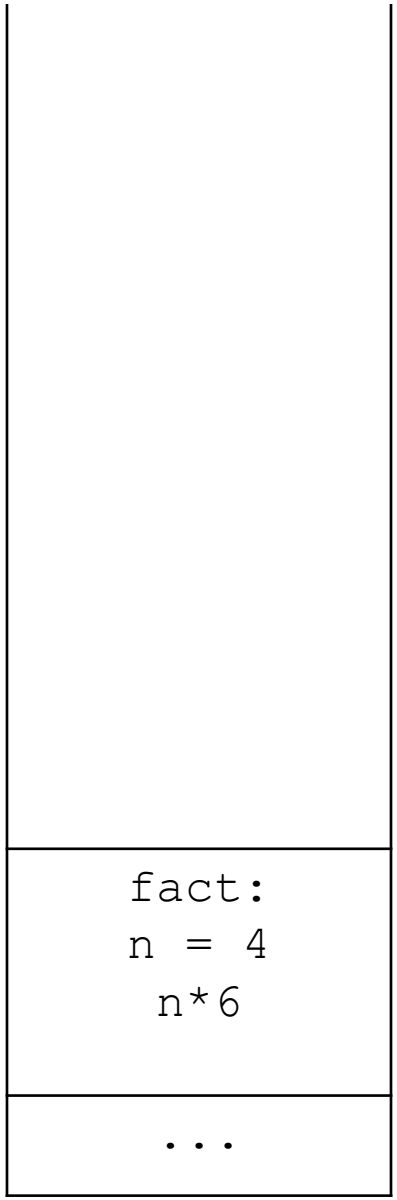


Recursive trace example

- Function that calls itself
- **Example:**

```
def fact(n):
  if n == 0:
    return 1
  else:
    return n*fact(n-1)
```

- Tracing `fact(4)`:

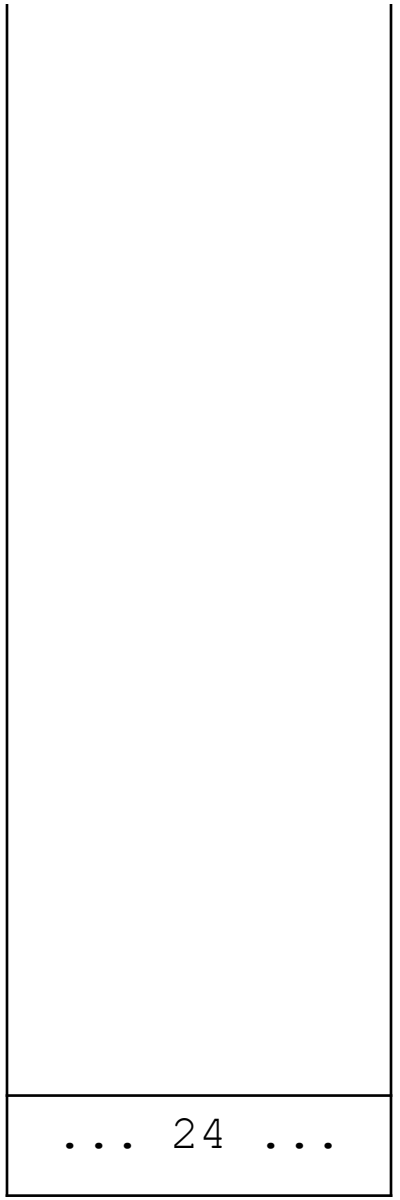


Recursive trace example

- Function that calls itself
- **Example:**

```
def fact(n):
  if n == 0:
    return 1
  else:
    return n*fact(n-1)
```

- Tracing `fact(4)`:



Anatomy of a recursive function

- Every recursive function has 2 main parts
- One or more *base cases*
  - Cases where we solve problem without recursion
- *Recursive case*
  - Uses recursion
  - Recursive calls must get closer to base case
- **Example:**

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)
```

Base case: 0

Recursive case

Closer to 0

Malformed recursion

- Consider the following function:

```
def fact(n):
    return n * fact(n-1)
```

- `fact(2)` calls `fact(1)`
- `fact(1)` calls `fact(0)`
- `fact(0)` calls `fact(-1)`
- `fact(-1)` calls `fact(-2)`
- ...and so on
- "Infinite recursion"
  - `RecursionError`: Python only allows you to call ~1000 functions at the same time

Different example

- Consider the following function:

```
def fact(n):
  if n == 0:
    return 1
  else:
    return (n-1) * fact(n) # Typo!
```

- `fact(2)` calls `fact(2)`
- `fact(2)` calls `fact(2)`
- ...and so on
- Recursive calls must always get closer to base case

Recursion example

- Fibonacci numbers
  - First two numbers are 1 and 1
  - Every other Fibonacci number is sum of previous two
- Formally, $F(n) = \begin{cases} 1, & \text{if } n = 1 \text{ or } 2 \\ F(n-1) + F(n-2), & \text{otherwise} \end{cases}$
- I'll write a Python function to calculate the `n`th Fibonacci number and print the first 10 Fibonacci numbers

Recursive design

- To solve a problem using recursion:
  - Solve just one step of problem
    - Use recursion to solve the rest
  - Solve tiny problems directly (base case)
    - 0 or 1 are common
- **Example:**
  - Write a recursive `list_sum` function to return the sum of a list
    - One step: add first/last element to sum
      - Recursively sum the rest
    - Base case: 0 or 1 element

```
def list_sum(list):
    if len(list) == 0:
        return 0
    else:
        return list[0] + list_sum(list[1:])
```

Recursive design exercise

- Write a recursive `is_palindrome` function that accepts a string and returns whether it is a *palindrome*
  - Same forwards and backwards
    - `huh`
    - `noon`
    - `racecar`

Coming up

- Object-oriented design
- Software testing

- **Homework 3** due next Friday (Sept 26)
- **Recommended reading:** week 4 in textbook