Outline

1. Comprehension Review
   - Lists

2. Function Arguments
   - Positional and Named Arguments
   - Variable Number of Arguments
   - Variables Declared Outside Function

3. Functional Programming
   - Background
   - Higher Order Functions
   - Partial Application
List Comprehensions

- \([\text{expr} \ \text{for} \ v \ \text{in} \ \text{iter}]\)
- \([\text{expr} \ \text{for} \ v_1,v_2 \ \text{in} \ \text{iter}]\)
- \([\text{expr} \ \text{for} \ v \ \text{in} \ \text{iter} \ \text{if} \ \text{cond}]\)
- Translate the following into a loop:

\[
\text{res} = [v_1 \times v_2 \ \text{for} \ v_1, v_2 \ \text{in} \ \text{lst} \ \text{if} \ v_1 > v_2]
\]
\[
\text{res} = []
\]
\[
\text{for} \ \{1\} \ \text{in} \ \{0\}:
\]
\[
\quad \text{if} \ \{2\}:
\]
\[
\quad \{3\}
List Comprehensions

- \([\text{expr} \ \text{for} \ v \ \text{in} \ \text{iter}]\)
- \([\text{expr} \ \text{for} \ v_1, v_2 \ \text{in} \ \text{iter}]\)
- \([\text{expr} \ \text{for} \ v \ \text{in} \ \text{iter} \ \text{if} \ \text{cond}]\)

Translate the following into a loop:

\[
\text{res} = [v_1 \times v_2 \ \text{for} \ v_1, v_2 \ \text{in} \ \text{lst} \ \text{if} \ v_1 > v_2]
\]
\[
\text{res} = []
\]
\[
\text{for} \ \{1\} \ \text{in} \ \text{lst}:
\]
\[
\text{if} \ \{2\}:
\]
\[
\{3\}
List Comprehensions

- [expr for v in iter]
- [expr for v1, v2 in iter]
- [expr for v in iter if cond]

Translate the following into a loop:

```python
res = [v1 * v2 for v1, v2 in lst if v1 > v2]
res = []
for v1, v2 in lst:
    if {2}:
        {3}
```
List Comprehensions

- \([\text{expr for v in iter}]\)
- \([\text{expr for v1,v2 in iter}]\)
- \([\text{expr for v in iter if cond}]\)

**Translate the following into a loop:**

```python
res = [v1 * v2 for v1, v2 in lst if v1 > v2]
res = []
for v1, v2 in lst:
    if v1 > v2:
        {3}
```
List Comprehensions

- `[expr for v in iter]
- `[expr for v1,v2 in iter]
- `[expr for v in iter if cond]

Translate the following into a loop:

```python
res = [v1 * v2 for v1, v2 in lst if v1 > v2]
res = []
for v1, v2 in lst:
    if v1 > v2:
        res.append(v1 * v2)
```
[[[4 for 3 in 2] for 1 in 0]]

Translate the following into the above list comprehension, given that lst2 is a list of tuples

```python
res = []
for tup in lst2:
    inter = []
    for x in tup:
        inter.append(x)
    res.append(inter)
```
[ [4 for 3 in 2] for 1 in lst2 ]

Translate the following into the above list comprehension, given that lst2 is a list of tuples

```python
res = []
for tup in lst2:
    inter = []
    for x in tup:
        inter.append(x)
    res.append(inter)
```
Nested List Comp

- \[[4 \text{ for } 3 \text{ in } 2] \text{ for } \text{tup in lst2}\]

- Translate the following into the above list comprehension, given that lst2 is a list of tuples

```python
res = []
for tup in lst2:
    inter = []
    for x in tup:
        inter.append(x)
    res.append(inter)
```
[4 for 3 in tup] for tup in lst2

Translate the following into the above list comprehension, given that lst2 is a list of tuples

res = []
for tup in lst2:
    inter = []
    for x in tup:
        inter.append(x)
    res.append(inter)
Nested List Comp

- \[ \{[4 \text{ for } x \text{ in } \text{tup}] \text{ for } \text{tup} \text{ in } \text{lst2}] \]

- Translate the following into the above list comprehension, given that lst2 is a list of tuples

```python
res = []
for tup in lst2:
    inter = []
    for x in tup:
        inter.append(x)
    res.append(inter)
```
Nested List Comp

- \[
  \[x \text{ for } x \text{ in } \text{tup} \text{ for } \text{tup in } \text{lst2}\]
- Translate the following into the above list comprehension, given that lst2 is a list of tuples

```python
res = []
for tup in lst2:
    inter = []
    for x in tup:
        inter.append(x)
    res.append(inter)
```
Extra 'for's and 'if's

[x for x in lst1 if x > 2 for y in lst2 for z in lst3 if x + y + z > 8]

Translation:

res = []
for x in lst1:
    if x > 2:
        for y in lst2:
            for z in lst3:
                if x + y + z > 8:
                    res.append(x)
Translate this??

```python
res = []
for x in lst1:
    if x > 2:
        for y in lst2:
            for z in lst3:
                if x + y + z > 8:
                    res.append(x)
```
Extra ‘for’s and ’if’s

- \[x \text{ for } x \text{ in } \text{lst1} \text{ if } x > 2 \text{ for } y \text{ in } \text{lst2} \text{ for } z \text{ in } \text{lst3} \text{ if } x + y + z > 8]\]

- Translate this??

```python
res = []
for x in lst1:
    if x > 2:
        for y in lst2:
            for z in lst3:
                if x + y + z > 8:
                    res.append(x)
```

Harry Smith  (University of Pennsylvania)  
CIS 192 Lecture 3  
September 13, 2017  16 / 39
Outline

1. Comprehension Review
   - Lists

2. Function Arguments
   - Positional and Named Arguments
   - Variable Number of Arguments
   - Variables Declared Outside Function

3. Functional Programming
   - Background
   - Higher Order Functions
   - Partial Application
Positional Arguments

- `def func(arg1, arg2, arg3):
  - arg1 arg2 and arg3 are positional arguments
  - When calling `func` exactly 3 arguments must be given
  - The order in the call determines which `arg` they are bound to
- `func(a, b, c)
  - The expressions `a`, `b`, `c` are evaluated before the call
  - The value of `a` is bound to `arg1` in the body of `func`
  - Likewise `b` to `arg2` and `c` to `arg3`
  - Calling a function with the wrong number of args gives a `TypeError`
Named Arguments

- After the positional args, named args are allowed

```python
def func(arg1, named1=val1, named2=val2):
    # named1 and named2 are variables usable in the body of func
    # val1 and val2 are default values for those variables.
    # Omitting named arguments in a call uses the default value

func(a, named2=b, named1=c)
```

- named arguments can be given out of order

```python
func(a, named2=b)
```

- The default value, \texttt{val1} will be bound to \texttt{named1}
Default Arguments

- Default arguments are evaluated when the function is defined.
- In all calls, the object that the expression evaluated to will be used.
- If the default is mutable, updates in one call effect following calls.
- `def func(a=[])` Will mutate the default on each call.
- `def func(a=None):
    if a is None:
        a = []`

- Use None as the default to avoid mutation.
Memoization

- Memoization is an optimization technique that stores results of function calls.
- The previously computed answers can be looked up on later calls.
- Use a dictionary default arg to store answers.
- `def func(arg, cache={})`:
- Store answers in `cache[arg] = ans`.
- Check for `arg in cache` before doing any work.
Outline

1. Comprehension Review
   - Lists

2. Function Arguments
   - Positional and Named Arguments
   - Variable Number of Arguments
   - Variables Declared Outside Function

3. Functional Programming
   - Background
   - Higher Order Functions
   - Partial Application
*args

- A variable number of positional arguments can be specified
- Use *args in between positional and named args
  - Could use any identifier but args is conventional
- `def func(arg1, *args, named=val)`
  - args is a tuple of 0 or more objects
- `func(a, b, c)`
  - arg1 = a, args = (b, c)
  - named gets the default object
Required Keyword Args

- Any args after `*args` are keyword args
- If there is no default value specified, they are required keyword args

```python
def func(*args, named):
    # named is a required keyword arg
```

- To specify required keyword args without allowing variable positional args use `*`

```python
def func(arg1, *, named)
    # named is a required kwarg
    # func must take exactly one pos arg and one kwarg
```
**kwargs

- A variable number of kwargs can be specified
- Use **kwargs at the end
  - Could use any identifier but kwargs is conventional
- `def func(arg1, *args, named=val, **kwargs)`
  - kwargs is a dictionary of strings to values
  - The keys of kwargs are the names of the keyword args
- `func(a, extra1=b, extra2=c)`
  - arg1 = a, args = tuple()
  - named gets the default object
  - kwargs = {'extra1': b, 'extra2': c}
/** in Function Definition

- `def(*args)` args is a tuple that can take 0 or more values
- `def(**kwargs)` kwargs is a dictionary that can take 0 or more key-value pairs
/** in Function Calls

- \texttt{func(*expr)}
  - \texttt{expr} is an iterable
  - It gets \texttt{unpacked} as the positional arguments of \texttt{func}
  - Equivalently
    \texttt{seq = list(expr); func(seq[0], seq[1], ...)}

- \texttt{func(**expr)}
  - \texttt{expr} is a dictionary of form \{’string’: val, ...\}
  - It gets \texttt{unpacked} as the keyword arguments of \texttt{func}
  - Equivalently \texttt{func(string=val, ...)}
Outline

1. Comprehension Review
   - Lists

2. Function Arguments
   - Positional and Named Arguments
   - Variable Number of Arguments
   - Variables Declared Outside Function

3. Functional Programming
   - Background
   - Higher Order Functions
   - Partial Application
Closures

A function that knows about variable defined outside the function

```python
a = 42

def func():
    print(a)

func is a closure because it knows about a
```

Closures are read-only in Python

```python
a = 42

def func():
    print(a)
    a += 1

UnboundLocalError: local variable ’a’ referenced before assignment
```
**global**

- **global** can circumvent read-only closures
- the **global** keyword declares certain variables in the current code block to reference the global scope

```python
a = 42
def func():
    global a
    print(a)
a += 1
```

- This does not raise an error
- Variables following **global** do not need to be bound already
Outline

1. Comprehension Review
   - Lists

2. Function Arguments
   - Positional and Named Arguments
   - Variable Number of Arguments
   - Variables Declared Outside Function

3. Functional Programming
   - Background
   - Higher Order Functions
   - Partial Application
Functional programming started with lambda(\(\lambda\)) calculus
  - Alternative to Turing machines for exploring computability
  - Expresses programs as functions operating on other functions

Functional programming attempts to make it easier to reason about program behavior
  - Mathematical interpretation of functions allows mathematical proofs

If data is immutable and there are no side-effects then functions always behave the same way

Python data is mutable and allows side-effects
  - Has some functional concepts
  - Not an ideal functional programming environment
Outline

1. Comprehension Review
   - Lists

2. Function Arguments
   - Positional and Named Arguments
   - Variable Number of Arguments
   - Variables Declared Outside Function

3. Functional Programming
   - Background
   - Higher Order Functions
   - Partial Application
First Class Functions

- A higher order function is a function that:
  - Takes a function as one of its inputs
  - Outputs a function
- You can use functions anywhere you would use a value
- Functions are immutable so you can use them as dictionary keys
- Functions can be the return value of another function
Anonymous functions are function objects without a name

lambda arg: ret is the same as
def <lambda>(arg):
    return ret

Lambdas can have the same arguments as regular functions
  - lambda arg, *args, named=val, **kwargs: ret

Lambdas must be one-liners and do not support annotations
Higher Order Functions

- The most common are `map`, `filter`, and `reduce` (foldL)
- `map(f, seq)` returns an iterator containing each element of `seq` but with `f` applied
- `filter(f, seq)` returns an iterator of the elements of `seq` where `bool(f(seq[i]))` is True
- `filter(None, seq)` is the same as `filter(lambda x: x, seq)`
- `reduce` must be imported. `from functools import reduce`
- `reduce(f, seq, base)`
  - Builds up result by calling `f` on elements of `seq` starting with `base`
  - `f(...f(f(base, seq[0]), seq[1]), ...)`
  - If `base` is not specified then the first argument is `seq[0]`
  - Calling `reduce` on an empty sequence is a TypeError
Many functions will accept another function as a kwarg

- `sorted(seq, key=f)`
  - `sorted` will call `f` on the elements to determine order
  - The elements in the resulting list will be the same objects in `seq`
  - Have the key return a tuple to sort multiple fields

- `min(seq, key=f)` and `max(seq, key=f)` behave similarly

This is a good spot for `lambda`
Outline

1. Comprehension Review
   - Lists

2. Function Arguments
   - Positional and Named Arguments
   - Variable Number of Arguments
   - Variables Declared Outside Function

3. Functional Programming
   - Background
   - Higher Order Functions
   - Partial Application
Partial application creates a new function by supplying an existing function with some of its arguments.

Say you have `add(x, y): x + y`.

You want `add_3(y): 3 + y`.

`add_3 = add(3)` raises a `TypeError`.

`add_3 = functools.partial(add, 3)`