Defining Python Functions

- Python functions can be defined, like C, with a fixed number of parameters

```
def polly(x, a, b, c):
    return a * x ** 2 + b * x + c
```

- functions can be called, like C, with ***positional*** arguments

```
>>>
>>> polly(3, 5, -3, 6)
42
```

- or with ***keyword*** arguments

```
>>> polly(a=5, c=6, b=-3, x=3)
42
```

Or with both ***positional*** and ***keyword*** arguments (keyword must follow positional)

```
>>> polly(3, c=6, b=-3, a=5)
42
```

- functions can restrict how they are called using special argument / and *

Default values for Functions Arguments

- default values can be specified for parameters

```python
def polly(x, a=1, b=2, c=0):
    return a * x ** 2 + b * x + c
```

- allowing functions to be called without specifying all parameters

```
>>> polly(3)
15
>>> polly(b=1, x=1)
2
```

- means you can add an extra parameter to a function without changing existing calls, by giving parameter default value

Mutable Default values are dangerous

- the default value is a single instance
- fine for immutable types: numbers, strings, ...
- unexpected results from mutable types: lists, dicts, ...

```python
def append_one(x = []):
    x.append(1)
    return x
```

```
>>> append_one()
[1]
>>> append_one()
[1, 1]
>>> append_one()
[1, 1, 1]
```

Mutable Default values - workaround

```python
def append_one(x = None):
    if x is None:
        x = []
    x.append(1)
    return x
```

```
>>> append_one()
[1]
>>> append_one()
[1]
>>> append_one()
[1]
```

Mutable Default values - workaround

```
def append_one(x = None):
    if x is None:
        x = []
    x.append(1)
    return x
```

```
>>> append_one()
[1]
>>> append_one()
[1]
>>> append_one()
[1]
```

Variable Numbers of Function Arguments

- packing/unpacking operators * and ** allow variable number of arguments.
    - Use * to pack positional arguments into tuple
    - Use ** to pack keyword arguments into dict

```python
def f(*args, **kwargs):
    print('positional arguments:', args)
    print('keywords arguments:', kwargs)
```

```
>>> f("COMP", 2041, 9044, answer=42, option=False)
positional arguments: ('COMP', 2041, 9044)
keywords arguments: {'answer': 42, 'option': False}
```

Packing Function Arguments

- * and ** can be used in reverse for function calls
  - Use * to unpack iterable (e.g. list or tuple) into positional arguments
  - Use ** to unpack dict into keyword arguments

```
>>> arguments = ['Hello', 'there', 'Andrew']
>>> keyword_argments = {'end' : '!!!\n', 'sep': ' --- '}
>>> print(arguments, keyword_argments)
['Hello', 'there', 'Andrew'] {'end': '!!!\n', 'sep': ' --- '}
>>> print(*arguments, **keyword_argments)
Hello --- there --- Andrew!!!
```

No main function

- Python has no special “main” function called to started execution (unlike e.g C)
- importing a file executes any code in it
- special global variable **`__name__`** set to module name during import
- if a file is executed rather than imported, **`__name__`** set to special value **`__main__`**
- so can call a function when a file is executed like this

```
if __name__ == '__main__':
    initial_function()
```

“

docstrings

- A Python Docstring is a string specified as first statement of function
- use ””” triple-quotes

```python
def polly(x, a, b, c):
    """calculate quadratic polynomial"""
    return a * x ** 2 + b * x + c
```

- provides documentation to human readers but also available for automated tools

```python
>>> polly.__doc__
'calculate quadratic polynomial'
```

```python
def polly(x, a, b, c):
    """calculate quadratic polynomial
    a -- squared component
    b -- linear component
    c -- offset
    """
    return a * x ** 2 + b * x + c
```

variable scope

- a variable assigned a value in a function is by default *__local__ to the function
- a variable not assigned a value in a function is by default *__global__ to entire program
- keyword **`global`** can be used to make variable global

variable scope - example

```python
def a():
    x = 1
    print('a', x, y, z)
def b():
    x = 2
    y = 2
    a()
    print('b', x, y, z)
def c():
    x = 3
    y = 3
    global z
    z = 3
    b()
    print('c', x, y, z)
```

source code for scope.py

```
>>> x = 4
>>> y = 4
>>> z = 4
>>> c()
a 1 4 3
b 2 2 3
c 3 3 3
```

nested functions

covered in later lecture

lambda

covered in later lecture