Python

CSE 216 : Programming Abstractions

Department of Computer Science Stony Brook University

Dr. Ritwik Banerjee

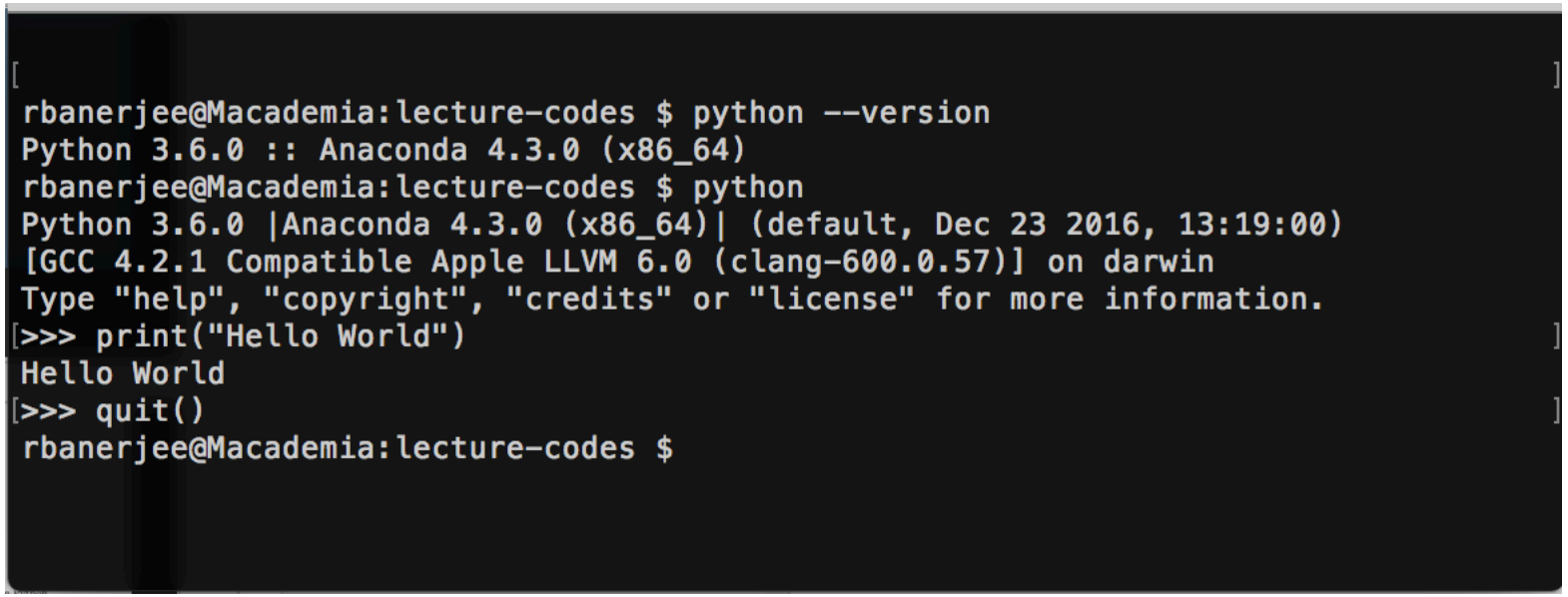
A Brief Background

- Developed in the late 1980s, primarily by Guido Van Rossum.
- Python 2.0 was released in 2000.
- Python 3.0 was released in 2008.
- Unlike Java, Python 3.x is *not* backward compatible.
  - That is, if you write a program in a version 2.x., it may not work on modern Python interpreters.
  - The official Python wiki says "Python 2.x is legacy, Python 3.x is the present and future of the language". So, we will use Python 3.x in this course.

- **For additional help**
  - The standard Python documentation (https://docs.python.org/3/) contains a tutorial.
  - The beginner's guide to Python (https://wiki.python.org/moin/BeginnersGuide)

Coding Environment

- Many computers come pre-installed with Python. If not, you can download and install it from https://www.python.org/downloads/.
- The Python interpreter can be launched straight-away from the terminal. This is an interactive shell, and perhaps the best way to get started!



Coding Environment

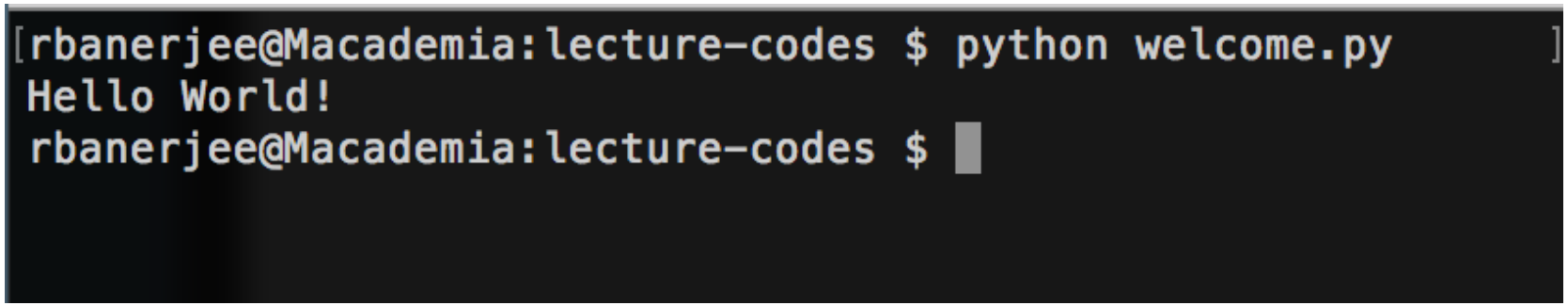
The Python installation also usually comes with a GUI environment called IDLE, which contains a Python shell.

For more complex projects, you may want to use a IDE like PyCharm or PyDev+Eclipse.

Running a simple program

```
# Welcome.py
# Display the one true message
# In case you haven't figured it out yet,
# the hash marks a comment line.
print("Hello World!")

# No semi-colon needed at the end of a statement
```



Syntax: blocks and indents

- Instead of brackets, Python uses indented block structures.
- So, instead of

```
if (x) {
    if (y) {
        a_subroutine();
    }
    another_subroutine();
}
```

- Python looks like:

```
if x:
    if y:
        a_subroutine()
    another_subroutine()
```

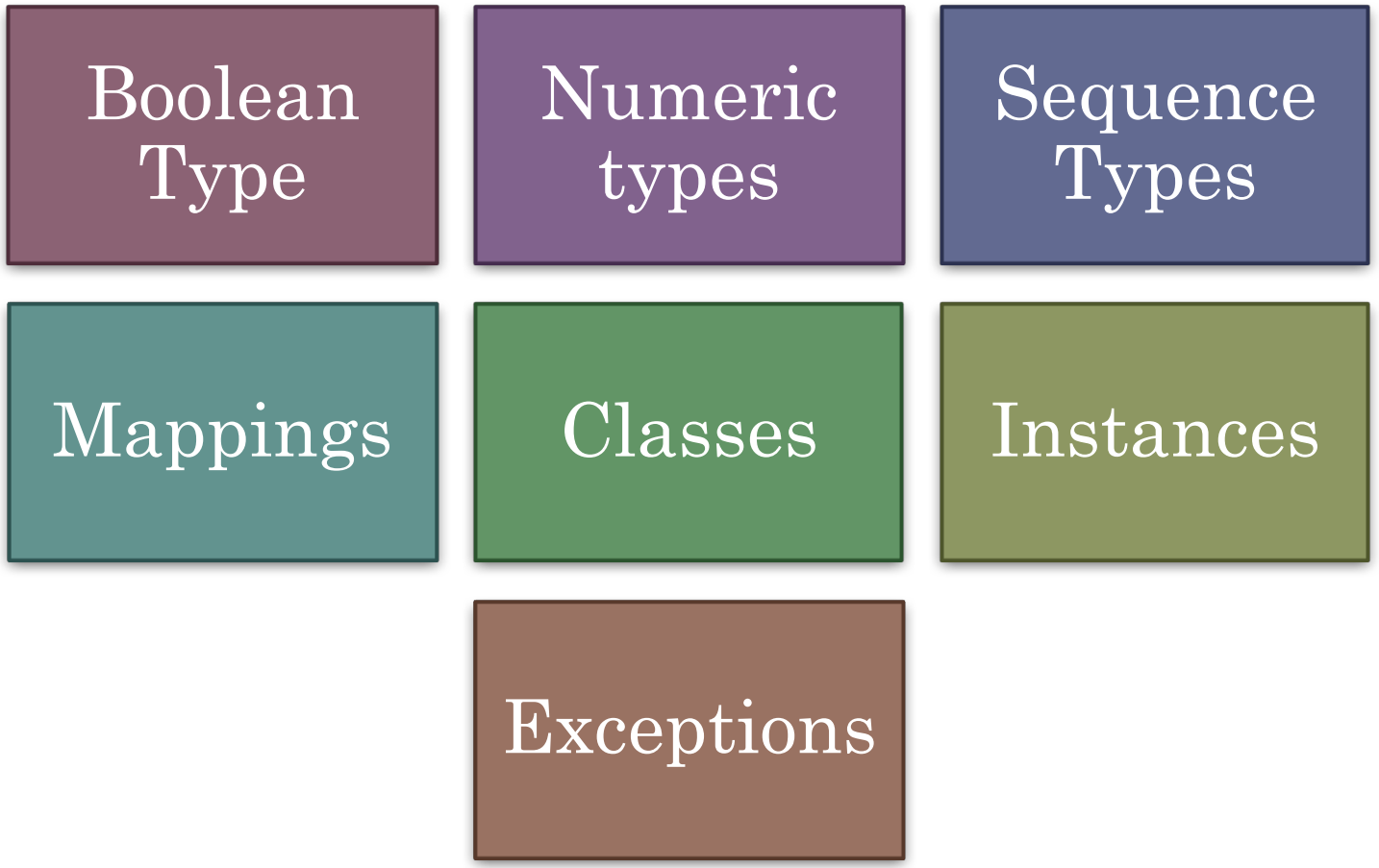
- The convention is to use 4 spaces (not tabs).
  - This is not a *formal* requirement, but using anything other than 4 spaces will make your code incompatible with most other python codes.

Objects

- Python is an object-oriented language.
- Everything – including numbers – is an object.
- Every object has an `id()` and a `type()`
  - The id remains unchanged from the creation until the destruction of the binding between the name and the object.
  - In the C implementation, the id is address of the object in memory.

```
>>> x = 3**3
>>> type(x)
<class 'int'>
>>> id(x)
4502620816
```

Built-in Types



Boolean Type

- An object is considered true by default, unless its class defines either a `__bool__()` method that returns `False` or a `__len__()` method that returns zero.
- The common built-in objects that are equivalent to false are
  a) constants defined to be false: `None` and `False`.
  b) zero of any numeric type
  c) empty sequences and collections
- The boolean operations are `and`, `or`, `not`.
  - With `or`, the second argument only gets evaluated if the first argument is true.
  - With `or`, the second argument only gets evaluated if the first argument is false.
- In Python, Booleans are a subtype of integers.
  - 0 is false, everything else is true.

Numeric Types

- Python has three numeric types (not considering Boolean):
  1. Integers
  2. Floating-point numbers
  3. Complex Numbers
- An `int` has unlimited precision, which is why there is no separate data type called 'long' in Python.
- A `float` are implemented as `double` in C.
- **A** `complex` is a complex number (*e.g.*, `5 + 3j`), implemented as two floating-point numbers denoting the real and imaginary .parts respectively

Type Conversion

Python performs the following type conversions for numbers:

- `int(4.2)` → `4`
- `float(4)` → `4.0`
- `round(4.51)` → `5`
- `round(4.5)` → `4`
- `int(4.2)` → `4`

Built-in functions for numeric types

The following functions use the math module:

```
>>> max(2,3,4)  # 4
>>> min(2,3,4)  # 2
>>> round(4.51) # 5
>>> abs(-3.2)   # 3.2
>>> pow(3,2)    # 9 (same as 3**2)
```

Sequence Types

Lists

Tuples

Ranges

Two additional sequence types:

- String
- Binary data

Common Sequence Operations

Operation	Result
`x in s`	`True` if an item of *s* is equal to *x*, else `False`
`x not in s`	`False` if an item of *s* is equal to *x*, else `True`
`s + t`	the concatenation of *s* and *t*
`s * n` or `n * s`	equivalent to adding *s* to itself *n* times
`s[i]`	*i*th item of *s*, origin 0
`s[i:j]`	slice of *s* from *i* to *j*
`s[i:j:k]`	slice of *s* from *i* to *j* with step *k*
`len(s)`	length of *s*
`min(s)`	smallest item of *s*
`max(s)`	largest item of *s*
`s.index(x[, i[, j]])`	index of the first occurrence of *x* in *s* (at or after index *i* and before index *j*)
`s.count(x)`	total number of occurrences of *x* in *s*

Source: https://docs.python.org/3/library/stdtypes.html#common-sequence-operations

Lists

- A **list** is a *mutable* sequence.
- It *can* be heterogeneous, but usually, programmers use homogeneous lists.
- List construction:

```
1. a_list = []       # create an empty list
2. a_list = list()   # same as line 1.
3. a_list = [2,3,4] # create a list with these three integers
4. a_list = ['red', 'green', 'blue'] # create a list with these
                                     # three strings
5. a_list = list('red') # create the list ['r', 'e', 'd']; this is
                        # possible because a string is a sequence
```

List methods

We are using the python syntax for method signature below, which is:

```
<method_name>(var_name: var_type, ...): return_type
```

- `append(x: object): None`
- `insert(index: int, x: object): None`
- `remove(x: object): None`
- `index(x: object) int`
- `count(x: object): int`
- `sort(): None`
- `reverse(): None`
- `extend(l: list): None`

```
>>> a_list = [2,3,4,1,16,30
>>> len(a_list)
6
>>> max(a_list)
30
>>> min(a_list)
1
>>> sum(a_list)
56
>>> all(a_list)
True
>>> any(a_list)
True
>>> sorted(a_list)
[1, 2, 3, 4, 16, 30]
```

```
>>> b_list = list()
>>> all(b_list)
True
>>> any(b_list)
False
>>> b_list.append('red')
>>> any(b_list)
True
>>> all(b_list)
True
```

Built-in Functions with Lists

```
>>> a = [1,2]
>>> b = [3,4]
>>> a + b
 [1, 2, 3, 4]
>>> c = a + b
>>> d = 2 * a
>>> d
 [1, 2, 1, 2]
>>> d = a * 2
>>> d
 [1, 2, 1, 2]
>>> e = d[2:4]
>>> e
 [1, 2]
>>> e = d[1:3]
>>> e
 [2, 1]
```

```
>>> a = [1,3,5,4,11,-2,0,25,-5]
>>> a[-1]
 -5
>>> a[-2]
 25
>>> 25 in a
 True
>>> 100 in a
 False
```

Operators on Lists

List Comparisons

```
>>> a = ["red", "green", "blue"]
>>> b = ["red", "blue", "green"]
>>> a == b
False
>>> a != b
True
>>> a is b
False
>>> a >= b
True
>>> b = a
>>> a is b
True
>>> a >= b
True
>>> a == b
True
```

Tuples

- A **tuple** is an immutable sequences, typically used to store collections of *heterogeneous* data.
  - We can think of them as the “record” data type in Python.
- **Constructing a tuple**

```
>>> a = (11, 22, 33, )
>>> b = 'aa', 'bb’
>>> c = 123,
>>> a
(11, 22, 33)
>>> b
('aa', 'bb')
>>> c
(123,)
>>> type(c)
<type 'tuple'>
```

Heterogeneous Tuples

- With heterogeneous tuples, it usually makes more sense to access the data using names instead of indices. To do this, use `collections.namedtuple()`.
  - This is *not* a built-in type.

```
>>> from collections import namedtuple
>>> Phone = namedtuple('Phone', ['owner', 'number'])
>>> p = Phone('John', '1231231231')
>>> p
Phone(owner='John', number='1231231231')
>>> p[0]
'John’
>>> p.owner
'John'
>>> p.number
'1231231231'
```

Heterogeneous Tuples

- With heterogeneous tuples, it usually makes more sense to access the data using names instead of indices. To do this, use `collections.namedtuple()`.
  - This is *not* a built-in type.

```
>>> q = Phone('Jack', 1231231231)
>>> q.number
1231231231
>>> owner, number = q
>>> owner
'Jack'
>>> number
1231231231
>>> tuple("apple")
('a', 'p', 'p', 'l', 'e')
```

Ranges

- The **range** data type represented immutable sequences of numbers.
- There are two ways of constructing a range:

1. *class* range(*stop*)

```
>>> range(10) # ‘start’ has a default value of 0
range(0, 10)
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

2. *class* range(*start*, *stop*[, step])

```
>>> list(range(0, 10, 3)) # ‘step’ has a default value of 1.
[0, 3, 6, 9]
>>> list(range(0, 10, -2))
[ ]
>>> list(range(0, -10, -2)) # count down with negative step
[0, -2, -4, -6, -8]
```

Ranges

- The **range** data type represented immutable sequences of numbers.
- What happens when you print a range?

```
> print(list(range(10)))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
> print(range(10))
range(0, 10)
```

- In many ways the object returned by `range()` behaves like a list, but it isn't.
- It is an object that returns the successive items of the desired sequence when you iterate over it, but it doesn't really keep the sequence pre-constructed.
  - Saves space.

Text Sequences – Strings

- A string in Python is an immutable sequence of characters.
  - To be more precise, an immutable sequence of Unicode code points.
  - Internally, it is implemented the same way as in C – a char array that is terminated by the `NULL` character '`\0`'.

```
s1 = 'a string can be defined in single quotes'
s2 = "or double quotes"
s3 = """for multiline string,
        three double quotes are required"""
s4 = '''three single quotes
        work as well'''
```

- In the Python shell, you can type `help(str)` to see the documentation.

Text Sequences - Strings

Constructing strings

```
>>> s1 = str()
>>> s2 = str("this is a
string")
>>> s3 = "this is a string"
>>> s1
''
>>> s2
'this is a string'
>>> s3
'this is a string'
```

Text Sequences - Strings

Sequence methods on strings

```
s = "this is a string"
>>> len(s)
16
>>> max(s)
't'
>>> min(s)
' '
>>> s + s
'this is a stringthis is a string'
>>> 2*s
'this is a stringthis is a string'
```

Text Sequences - Strings

Sequence methods on strings

```
s = "this is a string"
>>> s[2:6]
'is i'
>>> s[:1]
't'
>>> s[3:]
's is a string'
>>> s[-1]
'g'
>>> '' in s
True
>>> 'balloon' in s
False
```

String Methods

- `str.capitalize()`
- `str.lower()`
- `str.endswith(`*`suffix`*`)`
- `str.find(`*`substring`*`)`
  - The `find()` method should be used only if you need to know the position of substring. Otherwise, `in` is sufficient.
- For a complete (or mostly complete) list of methods to do string processing:
  - https://docs.python.org/3/library/text.html#textservices

Sets and Dictionaries

- A **set** is an *unordered* collection of distinct elements*.
- Two built-in set data types:
  1. `set` – mutable; we can add() and remove() elements
  2. `frozenset` – immutable
- The mapping data type is called a **dictionary**, which maintains key-values pairs of arbitrary types*.

```
s = {"apple", "banana", "mango"}
print(type(s))

a = dict(one=1, two=2, three=3)
b = {'one': 1, 'two': 2, 'three': 3}
c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
d = dict([('two', 2), ('one', 1), ('three', 3)])
e = dict({'three': 3, 'one': 1, 'two': 2})
print(a == b == c == d == e)
```

```
<class 'set'>
True
```

* Technically, these elements are hashable objects, but we will look into that when we study the object-oriented data model of Python in detail.

Dictionary operations

```
a = dict(one=1, two=2, three=3)
print(a.keys())
print(list(a.keys()))
print(list(a.values()))

a['four'] = 4 # set key to value
a['four']     # return the value with this key
a['five']     # throw KeyError if key is not in dictionary
```

```
dict_keys(['one', 'two', 'three'])
['one', 'two', 'three']
[1, 2, 3]
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
4
Traceback (most recent call last):
  File "main.py", line 12, in <module>
    a['five']       # throw KeyError if key is not in dictionary
KeyError: 'five'
```

Control Flow

```
from math import pi

if radius >= 0:
    area = radius * radius * pi
    print("The area of a circle with radius ", radius, "is
", area)
else:
    print("I think that was an invalid in put")
```

Control Flow

```
if score >= 90.0:
    grade = 'A'
else:
    if score >= 80.0:
        grade = 'B'
    else:
        if score >= 70.0:
            grade = 'C'
        else:
            if score >= 60.0:
                grade = 'D'
            else:
                grade = 'F'
```

```
if score >= 90.0:
    grade = 'A'
elif score >= 80.0:
    grade = 'B'
elif score >= 70.0:
    grade = 'C'
elif score >= 60.0:
    grade = 'D'
else:
    grade = 'F'
```

Control Flow

```
words = ['cat', 'window', 'defenestrate']

for w in words:
    print(w, len(w))

for i in range(10):
    print(i**2)

a = ['Mary', 'had', 'a', 'little', 'lamb']

for i in range(len(a)):
    print(i, a[i])
```

Control Flow

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(n, 'equals', x, '*', n//x)
            break
        else:
            print(n, 'is a prime number')
```

```
3 is a prime number
4 equals 2 * 2
5 is a prime number
5 is a prime number
5 is a prime number
6 equals 2 * 3
7 is a prime number
7 is a prime number
7 is a prime number
7 is a prime number
7 is a prime number
8 equals 2 * 4
9 is a prime number
9 equals 3 * 3
```

Control Flow

```
for num in range(2, 10):
    if num % 2 == 0:
        print("Found an even number", num)
        continue
    print("Found a number", num)
```

```
Found an even number 2
Found a number 3
Found an even number 4
Found a number 5
Found an even number 6
Found a number 7
Found an even number 8
Found a number 9
```

Control Flow

- There is a special statement in Python called **pass**:
- The `pass` statement does nothing, and can be used when a statement is required syntactically but the program requires no action.
- Example:

```
while True:
    pass
```

Functions

- The keyword **def** introduces a function definition.
- Function name
- List of formal parameters
- The first statement can (optionally) be a string literal.
  - Python's way of providing the documentation string (**docstring**).

```
def fibseries(n):
    "Print a Fibonacci series up to n."
    a, b = 0, 1
    while a < n:
        print(a, end = ' ')
        a, b = b, a + b
    print()
```

Functions

Multiple assignments:

- The variables ‘a’ and ‘b’ simultaneously get the values 0 and 1.
- In the last line of the loop, we use multiple assignments again.
  - **! The expressions on the right-hand side are evaluated before any assignment takes place.**
  - **! The right-hand side expressions are evaluated left-to-right.**
- Keep in mind that the comma is implicitly a tuple-construction:
  - `b, a + b` is the tuple `(b, a + b)`

```
def fibseries(n):
    "Print a Fibonacci series up to n."
    a, b = 0, 1
    while a < n:
        print(a, end = ' ')
        a, b = b, a + b
    print()
```

Functions

What does the function `fibseries` return?

- Even functions without a `return` statement do return a value: `None`.
- This is the null value in Python, much like `void` in Java. It is a reserved keyword.
- As an example, here is a function that returns the integer type.

```
def fib(n):
    "Print the nth Fibonacci number."
    if n < 0:
        print("Fibonacci series begins with n = 0")
    elif n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Variable number of parameters

Specify a default value for one or more arguments:

```
def confirmation(query,
                 max_attempts=3,
                 reminder='Please try again with a valid input.'):
    while True:
        response = input(query)
        if response in ('y', 'Y', 'yes', 'Yes', 'YES'):
            return True
        if response in ('n', 'N', 'no', 'No', 'NO'):
            return False
        max_attempts = max_attempts - 1
        if max_attempts < 0:
            raise ValueError('Too many invalid responses.')
        print(reminder)
```

Variable number of parameters

Specify a default value for one or more arguments:

```
> confirmation("Are you sure? ")
Are you sure? No
False
> confirmation("Are you sure? ", 1)
Are you sure? ummm...
Please try again with a valid input.
Are you sure? I'm not sure
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "main.py", line 33, in confirmation
    raise ValueError('Too many invalid responses.')
ValueError: Too many invalid responses.
> confirmation("Are you sure? ", 1, "Please enter [y]es/[n]o.")
Are you sure? not really
Please enter [y]es/[n]o.
Are you sure? y
True
```

```
> i = 0
> def default_value_test(arg = i):
...     print(arg)
...
> default_value_test()
0
> default_value_test(5)
5
```

How are default values evaluated?

Default values are evaluated *at the point of a function definition.*

```
> def default_value_test(num, a_list = list()):
...     a_list.append(num)
...     return a_list
...
> default_value_test(1)
[1]
> default_value_test(2)
[1, 2]
> default_value_test(3)
[1, 2, 3]
> default_value_test(4)
[1, 2, 3, 4]
```

How are default values evaluated?

- The default value is *evaluated only once*.
- This is very important for *mutable* objects like lists, dictionaries, etc.

```
> def default_value_test(num, a_list = None):
...     if a_list is None:
...       a_list = list()
...     a_list.append(num)
...     return a_list
...
> default_value_test(1)
[1]
> default_value_test(2)
[2]
> default_value_test(3)
[3]
```

How are default values evaluated?

To avoid such sharing of the default value between subsequent function calls, rewrite the function like this.

Keyword and positional arguments

An **argument** is a value passed to a function when calling the function.

Python provides two kinds of arguments:

- A **keyword argument** (`kwarg`) is preceded by an identifier in a function call.
- E.g., construct a complex number by calling `complex(real=3, imag=2)`.
- A **positional argument** is an argument that is *not* a keyword argument. These can appear at the beginning of an argument list.
- E.g., construct a complex number by calling `complex(3, 2)`.

Variable number of parameters

Keyword Arguments

```
def confirmation(query,
                 max_attempts=3,
                 reminder='Please try again with a valid input.')
```

- This function can be called in any of the following ways:
  - `confirmation(query='Are you sure? ')`
  - `confirmation(query='Are you sure? ', max_attempts=2)`
  - `confirmation(query='Are you sure? ',`
    `max_attempts=2,`
    `reminder='Try again.')`
  - `confirmation(query='Are you sure? ', reminder='Try again.')`

Variable number of parameters

Keyword Arguments

```
def confirmation(query,
                 max_attempts=3,
                 reminder='Please try again with a valid input.')
```

- But the following are invalid:
  - `confirmation() # required arg missing`
  - `confirmation(query='Are you sure? ', 2) # non-keyword arg after kw arg`

Variable number of parameters

- Keyword arguments must follow positional arguments.
- All the keyword arguments passed must match one of the arguments accepted by the function, and their order is not important.
- A required argument may also be provided with a keyword.
- No argument may receive a value more than once.

Arbitrary number of arguments

- A function can be called with an arbitrary number of arguments.
- These arguments will be wrapped up in a tuple.
- Normal arguments may occur before the variable number of arguments.
  - They effectively "scoop up" all the remaining arguments passed to the function.
  - Often called **variadic** arguments.

```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

  - Any formal parameter after the `*args` must be keyword arguments.

```
def concat(*args, sep=' '):
    return sep.join(args)
```

Arbitrary number of arguments

- A dictionary can be passed as a formal parameter in the form **name.

```
def cheeseshop(kind, *arguments, **keywords):
    print("-- Do you have any", kind, "?")
    print("-- I'm sorry, we're all out of", kind)
    for arg in arguments:
        print(arg)
    print("-" * 40)
    for kw in keywords:
        print(kw, ":", keywords[kw])
```

```
> cheeseshop("Limburger", "It's very runny, sir.",
...            "It's really very, VERY runny, sir.",
...            shopkeeper="Michael Palin",
...            client="John Cleese",
...            sketch="Cheese Shop Sketch")
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
----------------------------------------
shopkeeper : Michael Palin
client : John Cleese
sketch : Cheese Shop Sketch
```

Function Annotations

- Even though Python is dynamically typed, it allows the programmer to include 'type hints' in the form of function annotations.

```
def sumtwo(a: int, b: int) -> int:
    return a + b
```

- The annotations are stored in the `__annotations__` attribute of the function as a dictionary and have no effect on any other part of the function.

```
> sumtwo.__annotations__
{'a': <class 'int'>, 'b': <class 'int'>, 'return': <class 'int'>}
```

PEPs and coding style

- Python programming guidelines and conventions are defined in **Python Enhancement Proposals**, or **PEP**s.
- PEP 8 has emerged as the nearly-universal style doctrine.
- Fortunately, most modern IDEs (e.g., PyCharm) abide by PEP 8 and will warn the programmer of any violations.
- For this course, we are not enforcing the style guide in its entirety, but we do require some minimal adherence, as described in this basic coding style guideline.