* Course Overview
  + Callables and lambdas
  + Extended forms of pythons syntax
  + Closures and decorators
  + Advanced use of comprehensions
* Review of Functions
  + Free functions
    - Functions defined at module scope
  + Methods
    - Functions defined within a class definition
  + Positional arguments are matched with formal arguments by position, in order
  + Keyword arguments are matched with formal arguments by name
  + Choice between the two is made at the call site
  + Arguments may have a default value
  + The default value for an argument is only evaluated once
  + Be careful when using mutable data type for default values
    - They may retain changes between calls
  + Functions are objects and can be passed around just like any other object
  + Python 3 system
  + Naming special functions
    - \_\_feature\_\_
    - “dunder” feature
    - Portmanteau of “double underscores”
* Callable Interface
  + \_\_call\_\_()
    - Allows instances of classes to be callable objects
    - \_\_call()\_\_ is invoked on objects when they are called like functions
  + Ex)
    - import socket
    - class Resolver:
    - def \_\_init\_\_(self):
    - self.\_cache = {}
    - def \_\_call\_\_(self, host):
    - if host not in self.\_cache:
    - self.\_cache[host] = socket.gethostbyname(host)
    - return self.\_cache[host]
    - resolve = Resolver()
    - resolve(‘sixty-north.com’) # return 93.93.131.30, syntax sugar for call below
    - resolve.\_\_call\_\_(‘sixty-north.com’) # return 93.93.131.30
  + since callable instances are just normal class instances, their classes can be define any other methods you want
* Classes Are Callable
  + Class objects and instances of classes are very different things
  + Class binds a class object to a named reference
  + Arguments passed to the class object are forwarded to the class’s \_\_init\_\_()
  + Classes are object factories
  + Classes produce new instances when they are invoked
  + cls
    - shortened version of “class”
    - very common in the python ecosystem
  + klass
    - deliberate misspelling of “class”
    - a bit more explicit
  + conditional expressions
    - evaluates to one of two expression depending on a Boolean
    - result = true\_value if condition else false\_value
* Lambdas
  + in many cases anonymous callable objects will suffice
  + lambda allows you to create such anonymous callable objects
  + use lambda with care to avoid creating inscrutable code
  + use greek letter lambda due to alonzo church’s work on the foundations of computer science
    - his lambda calculus forms the basis for many modern functional languages
  + sorted(iterable, key)
    - iterable like a list of names
    - key: must be a callable like a lambda
  + ex)
    - scientists = [‘Marie Curie’, ‘Albert Einstein’, ‘Rosalind Franklin’ …]
    - sorted(scientists, key=lambda name: name.split()[-1])
  + functions
    - def name(args): body
    - def is a statement which defines a function and binds it to a name
    - must have a name
    - arguments delimited by parentheses, separated by commas
    - zero or more arguments supported - zero arguments == empty parentheses
    - body is an indented block of statements
    - a return statement is required to return anything other than None
    - regular functions can have docstrings
  + lambda
    - lambda args: expr
    - expression which evaluate to a function
    - anonymous
    - argument list terminated by a colon, separated by commas
    - zero or more arguments supported - zero arguments == lambda
    - body is a single expression
    - the return value is given by the body expression, no return statement is permitted
    - lambdas cannot have docstrings
  + detecting callable objects
    - use callable function
  + ex)
    - def is\_even(x):
    - return x % 2 == 0
    - callable(is\_even) #True
    - is\_odd = lambda x: x % 2 == 1
    - callable(is\_odd) #True
* Summary
  + reviewed basics of python functions
  + use \_\_call\_\_() to make callable instances
  + associate state with callable objects
  + classes are callable objects
  + calling a class object creates an instance of the class
  + Lambdas are unnamed callable objects
  + when to use lambdas vs functions and other callables
  + use callable() t o determine if an object is callable
  + conditional expressions are a concise form of conditionals
  + classes are objects
* Extended Formal Argument Syntax
  + arbitrary numbers of positional arguments
  + arbitrary keyword arguments
  + positional only and keyword only arguments
  + extended call syntax
  + forwarding arbitrary function arguments
  + extended argument syntax
    - print(), print(“one”), print(“one”, “two”), etc
  + def function(\*args)
    - special argument syntax
    - called star args
    - positional arguments are passed in as a tuple
  + when you need to accept variable number of arguments with a positive lower bound use positional arguments with star argos
    - ex) def volume(length, \*lengths)
  + Rules for \*args
    - must come after normal positional arguments
    - must have only one \*args
    - only collections positional arguments
* Keyword and Positional-only Argu…
  + arbitrary keyword arguments
  + prefix argument with \*\* to accept arbitrary keyword arguments
  + conventionally called \*\*kwargs
  + ex)
    - def tag(name, \*\*kwargs)
    - tag(“img”, src=”Monet.jpg”, alt=”Sunrise)
    - the keyword arguments are passed in as a dictionary
  + can have variable positional arguments and variable keyword arguments
    - \*args must precede \*\*kwargs
    - any arguments before \*args must be positional
    - and arguments after \*args and before \*\*kwargs must be keyword
  + ex)
    - def print\_args(arg1, arg2, \*args, kwargs1, kwarg2)
  + can have positional arguments and variable keyword arguments
    - use \* without a name to bypass variable variable positional arguments
  + ex)
    - def name\_tag(first\_name, last\_name, \*, title=’’)
  + \*\*kwargs must be last in argument list
  + ex)
    - def print\_args(arg1, arg2, \*args, kwarg1, kwarg2, \*\*kwargs)
  + postionial only argument
    - include forward slash in arguments list
    - parity with modules implemented in other languages
    - prevent formal arguments names from becoming part of the API
    - prevent dependencies on the names
  + ex)
    - def number\_length(x, /)
  + extend arguments syntax applies to all types of callables
* positional arguments vs keyword arguments
* tuple
* Extended Call Syntax
  + Ex
    - def print\_args(arg1, arg2, \*args):
    - ….
    - t = (11, 12, 13, 14)
    - print\_args(\*t)
  + \*t unpacks the series into positional arguments
  + Can use \*\*kwargs to unpack a mapping type into keyword arguments
  + Ex)
    - def color(red, green, blue, \*\*kwargs):
    - ….
    - k = {‘red’: 21, ‘green’:68, ‘blue’:120}
    - color(\*\*k)
  + dict() uses \*\*kwargs in its initializer
  + argument forwarding
  + ex)
    - def trace(f, \*args, \*\*kwargs):
    - …
    - result = f(\*args, \*\*kwargs)
    - …..
* Summary
  + Extended argument syntax for accepting arbitrary positional arguments
  + As well as arbitrary keyword arguments
  + Specifying keyword-only arguments
  + Specifying positional-only arguments
  + Extended call syntax
  + Perfect argument forwarding
* Local Functions
  + def
    - used to define new functions
    - binds a function object to a name
    - executed at runtime
  + you can define functions inside other functions, called local functions
  + local functions
    - defined when def is executed
    - new copy made for each enclosing invocation
    - separate name binding each time
  + scoping rules
    - local
    - enclosing
    - global
    - built-int
  + name resolution for local functions starts with names in the local function
    - then checks enclosing scopes
    - finally module-level and built in name are checked
  + local functions are not members of their enclosing functions
  + defining one-off functions close to their use
  + similar to lambdas
  + more general than lambdas
  + these instance can be returned from the enclosing function
  + first-class functions
    - functions can be passed to and returned from functions
    - more generally, they can be treated like any other data
* Closures and Nested Scopes
  + How does a returned local function retain access to its enclosing scope
    - Once the local function is returned, the enclosing scope is gone
  + Local functions forms a closure
    - Closure records objects from enclosing scopes
    - Keeps recorded objects alive for use after the enclosing scope is gone
    - Implemented with the \_\_closure\_\_ attribute
  + A common use of closure is function factories
    - Functions that return other functions where it has been specialized in some way based on arguments to factories
    - Returned functions use both their own arguments as well as arguments to the factory
    - Combination of runtime functions definition and closures
  + Ex)
    - def raise\_to(exp):
    - def raise\_to\_exp(x):
    - return pow(x, exp)
    - return raise\_to\_exp
* The Nonlocal Keyword
  + Global
    - Introduces bindings from the global scope into another scope
  + We can bind the global message into local()
  + How can we bind names from enclosing scopes?
  + nonlocal keyword
    - insert a name binding from an enclosing scope into the local namespace
    - searches enclosing scopes from innermost to outermost
    - uses the first match found
  + ex
    - message = ‘global’
    - def enclosing():
    - message =’enclosing’
    - def local():
    - nonlocal message
    - message = ‘local’
    - print(‘message:’, message) #prints enclosing
    - local()
    - print(‘message:’, message) #prints local
    - enclosing()
    - print(‘message:’, message) #prints global
  + nonlocal will throw error if there are not matching enclosing binding
* Summary
  + Defining local functions
  + Enclosing scopes
  + Local functions can be returned
  + Factory functions
  + Closures keep enclosing-scope objects alive
  + nonlocal lets you bind names from enclosing scopes
  + Closures can associate state with a function
* Functions Decorators
  + Modify or enhance an existing function in a non-intrusive and maintainable way
  + Implemented as a callable that accepts a callable and returns a callable
  + A function accepting a function and returning a function
  + Syntax
    - @my\_decorator #applies decorators to functions
    - def my\_function():
    - …
  + Python decorator steps
    - Python compiles the base functions, creating a functions object
    - Python then passes functions object to decorator
    - New function is then bind to origin function
  + Decorators allow you to modify existing functions without changing their definition
  + Callers don’t need to change when decorators are applied
  + Decorator example
    - def escape\_unicode(f):
    - def wrap(\*args, \*\*kwargs):
    - x = f(\*args, \*\*kwargs)
    - return ascii(x)
    - return wrap
    - @escape\_unicode
    - def northern\_city():
    - return ‘Trmoso’
  + Decorators take a callable argument and return a callable
  + The callable we return is the local function wrap()
  + Wrap() uses a closure to access f after escape\_unicode() returns
* What Can Be a Decorator?
  + Classes as decorators
    - Classes are callable objects
    - Functions decorated with a class are replaced by an instance of the class
    - These instance must themselves be callable
  + We can decorate with a class as long as instances of the class implement \_\_call\_\_()
  + Example
    - class CallCount:
    - def \_\_init\_\_(self, f):
    - self.f = f
    - self.count = 0
    - def \_\_call\_\_(self, \* args, \*\*kwargs):
    - self.count += 1
    - return self.f(\*args, \*\*kwargs)
    - @CallCount
    - def hello(name):
    - print(…)
  + Instances as decorators
    - Python calls an instance’s \_\_call\_\_() when it’s used as a decorator
    - \_\_call\_\_()’s return value is used as the new function
    - Create groups of callables that you can dynamically control as a group
  + Example
    - class Trace:
    - def \_\_init\_\_(self):
    - self.enabled = True
    - def \_\_call\_\_(self, f):
    - def wrap(\*args, \*\*kwargs):
    - if self.enabled:
    - print(‘Calling {}’.format(f))
    - return f(\*args, \*\*kwargs)
    - return wrap
    - tracer = Trace()
    - @tracer
    - def rotate\_list(l):
    - …..
    - If we then set tracer.enabled = False, it will no longer print
  + Power and flexibility
    - The available callable objects give you a lot of flexibility when implementing decorators
  + Design space
    - Deciding which to use will depend a great deal upon what you’re trying to accomplish
* Applying Multiple Decorators
  + Ex)
    - @decorator1
    - @decorator2
    - @decorator3
    - def some\_function():  
       ….
  + The decorators are processed in reversed order
    - Functions passed to decorator3
    - Callable returned from decorator3 is passed to decorator2 and so on
  + Decorating methods is valid
* Preserving Function Metadata
  + by replace a functions with a callable, we lost important metadata from the original callable
    - things like \_\_name\_\_, \_\_doc\_\_
  + we can copy \_\_name\_\_ and \_\_doc\_\_ from our wrapped functions to our wrapper function
  + ex
    - def noop(f):
    - def noop\_wrapper():  
       return f()
    - noop\_wrapper.\_\_name\_\_ = f.\_\_name\_\_
    - noop\_wrapper.\_\_doc\_\_ = f.\_\_doc\_\_
    - return noop\_wrapper
  + functools.wrap()
    - replace decorator metadata with that of the decorated callable
    - it is a decorator that you apply to your wrapper function
* Parameterized Decorators
  + ex)
  + def check\_non\_negative(index):
  + def validator(f):
  + def wrap(\*args):
  + if args[index] < 0:
  + raise ValueError( ‘ adsfasd’)
  + return f(\*args)
  + return wrap
  + return validator
  + @check\_non\_negative(1)
  + def create\_list(value, size):
  + return [value] \* size
  + check\_non\_negative is not a decorator
  + a decorator takes a callable object as an argument and returns a callable
  + the return value of check\_non\_negative is the decorator
  + python takes the return value and passes function into it
  + the wrap function forms a closure over f and index
  + decorators are a powerful tool in python programming
  + decorators can be overused
  + use them when they improve maintainability, add clarity, and simplify your code
  + decorators modify existing callables extrinsically
  + apply decorators with “@”
  + decorators are callables that accept a callable and return a callable
  + decorators can be implement with functions, classes, or instances
  + apply multiple decorators to a callable
  + recursive application of basic rules
* Map