* Course Overview
* Overview
  + Install python
  + Write basic code
* Installing python
  + Install on windows
    - Python.org
    - Download
    - Add python 3.7 to path variable
    - Use windows powershell
    - Type ‘python’
  + Installation on macos
    - Python.org
    - Download for mac os
  + Installation on linux
    - Default to recent version of ubuntu
* Interactive Python
  + REPL(read evaluate print loop)
  + Type ‘\_’ to refer to most recently printed value
    - Can be used in expressions
    - Only in REPL
  + Type ‘print(‘string’) to print to command line, new line per print
  + In python 2 type ‘print “string”’
    - Is a function call
  + In python 3 type ‘print(‘string’)
  + Type ‘exit()’ to exit REPL
  + Type ‘Ctrl-Z + Enter’ to exit REPL on windows
  + Type ‘Ctrl-D’ to exit REPL on linux/macOS
* Significant Whitespace
  + Control flow section are terminated by :, which requires a body
  + Ex)
    - for i in range(5):
    - x = i \* 10
    - print(x)
  + Leading whitespace is syntactically significant
  + Python uses indentation levels to demarked code blocks
    - Four spaces for each level
    - Use empty line to terminated block
  + range(num) is from 0 to num - 1
  + requires readable code
  + no clutter
  + human and computer can’t get out of sync
  + Whitespace rules
    - Prefer four spaces
    - Spaces over tabs
    - Never mix spaces and tabs
    - Be consistent on consecutive lines
    - Only deviate to improve readability
* Python Culture
  + In REPL type ‘import this’ to show zen of python
* Using the Standard Library
  + Use the import keyword
  + Ex)
    - import [module]
  + use ‘module\_name.attribute\_name’ to access attribute
  + ex)
    - math.sqrt(81)
  + type help(object) to see attributes
  + ex)
    - help(math)
  + they ‘q’ to return to REPL
  + type help(math.factorial) to gain more info function
  + type ‘from math import factorial’ to just bring in the function
  + ex usage) factorial(n)
  + can also rename the function
  + ex) from math import factorial as fac
    - fac(10)
  + / is floating point division
    - 5 / 2 return 2.5
  + // is integer division
    - 5 / 2 returns 2
  + len(string) to return count of char in string
  + str(other type) to convert to string
* Summary
* Overview
  + Scalar types
  + Relational operators
  + Flow-control
  + Scalar types
    - Integers, collections(dictionaries) etc
    - Int, flot, None(null object), True, False
  + int
    - unlimited precision signed integer
    - 0b[num] for binary
    - Ex) 0b10 -> 2
    - 0o[num] for octal
    - 0x[num] for hexadecimal
    - int(num) to convert num to integer
    - int(num, base) to convert to base integer
  + float
    - IEEE-754 double-precision with 53 bits of binary precision
    - 15-16 significant digits in decimal
    - [num]e[num of trailing zeros]
    - Ex) 3e8 = 300000000.0
    - float(num) to convert to float
  + nan: not a number
  + inf: positive infinite number
  + -inf: negative infinite number
  + int and float results in float
  + None: null value
    - Often represents the absence of a value
    - REPL never prints none results
    - Can be assigned to variables
  + Us ‘is’ to check for type, returns bool True or False
  + Ex) a is None
  + bool
    - Boolean logical values
    - True, False
    - 0 is falsey, every other numb is truthy
    - Empty list is falsey, non empty list is truthy
    - Empty string is falsey, non empty string is truthy
    - “False” string is truthy, “True” string is truthy
* Relational Operators
  + ==: value equality / equivalence
  + !=: value inequality / inequivalence
  + <: less-than
  + >: greater-than
  + <=: less-than or equal
  + >=: greater-than or equal
* Control Flow
  + Conditional statement branch execution based on the value of an expression
  + if expression:
  + block
  + elif:
  + block
  + else:
  + block
  + elif is else if
* While-loops
  + while expression:
  + block
  + Augmented assignment operators
    - -=
    - +=
    - \*=
    - Etc
  + break: many languages support a loop ending in a predicate test
    - do while loop
  + python requires you to use while True and break
  + break jumps out of the inner-most executing loop to the line immediately after it
  + input() takes input from user
* Summary
  + scalar types: int, float, None, bool
  + if-elif-else
  + Control-C generates a KeyboardInterrupt exception
* Overview
  + str, bytes, list, and dict
  + for-loop
  + collections types
    - str, bytes, list, dict,
* String
  + str: data type for strings in python
  + sequence of Unicode code points
  + Immutable: cant modify its content after creation
  + Strings are enclosed in ‘ ’ or “ “
    - You must be consistent
  + Use one or the other to escape ‘ or “
  + Ex) “It’s a good thing.”
  + Ex) ‘ “Yes!”, he said, “I agree!” ‘
* String Literals
  + Adjacent literal strings are concatenated by compiler to single string
  + Ex) “first” “second” -> ‘firstsecond’
  + String with Newlines
    - Multiline strings: spread the literal across multiple lines
    - Escape sequences: Embed escape sequences in a single-line literal
  + Multiline Strings: use “ “ “ or ‘ ‘ ‘
    - Use “””
    - Ex)
    - “”” This is
    - A multiline
    - String “””
  + Escape sequences: put ‘\n’ into strings
  + Ex) ‘This string\nspans multiple\nlines’
  + New lines and operating systesm
    - Windows: carriage-return, line-feed(\r\n)
    - Linux and macOS carriage-return(\r)
  + Pythons translate \n to the appropriate newline sequence for your platform
    - Universal newlines
  + Escape Sequences
    - “This is a \” in a string “ -> ‘This is a “ in a string’
    - ‘This is a \’ in a string’ -> “This is a ‘ in a string”
    - ‘This is a \” and a \’ in a string’ -> ‘This is a “ and a \’ in a string’
    - Use \\ to escape \
    - Ex)
    - Print(‘A \\ in a string’) -> A \ in a string
  + Raw strings: what you see is what you get
    - Prefix with r
    - Ex) prefix = r’C:\Users\Merlin\Documents\Spells’
  + str(parameter) to convert parameter to string
  + strings are sequence types
    - can use square brackets to access individual characters of string
    - ex) s = ‘parrot’
    - s[4] -> ‘o’
  + in python there is no separate character type, s[4] returns a string
  + type(parameter) to return type of parameter
  + help(str) to get methods and properties on string
  + str.capitalize() returns a new string with first character capitalized
  + ex)
    - c = ‘oslo’
    - c.captialize()
  + str is Unicode
  + python 3 source encoding is UTF-8
* Bytes
  + Data type for sequences of bytes
  + Raw binary data
  + Fixed-width single-byte encodings
  + Prefix with b
  + Ex)
    - b'data’
    - b”data”
  + support the same operations as string
  + bytes.split() split by be ‘delimiter’, empty for spaces
  + converting between strings and bytes
    - must know the encoding to turn string to bytes
    - decoding for byte to string
  + ex)
    - norks = “….”
    - data = norks.encode(‘utf8’)
    - norwegian = data.decode(‘utf8’)
* Lists
  + Sequences of objects
  + Mutable
  + A workhorse in python
  + [] is used to defined lists
  + Ex)
    - [1,9,8]
  + Use square brackets to access data
  + Can have different types of objects in the same list
  + list.append() to add to end of list
  + list() to use list constructor
* Dict
  + Fundamental data structure
  + Maps keys to values
  + Known as maps or associative arrays
  + {} to defined lists
  + { key: value}
  + Use [key] to access elements
  + If you use key that has not been added, a new entry will be added
  + Since python 3.7 entries are keep in insertion order
* For-loop
  + Visit each item in an iterable sequence
  + Ex)
    - for item in iterable:
  + body
  + For dictionary you get the keys
  + Ex)
    - for color in colors:
    - print(color, colors[color])
* Putting it all Together
  + Http data is provided by bytes
  + REPL is good for short-lived work and experimentation
  + Use an editor/IDE for larger or longer-lived projects
* Summary
* Overview
  + Reusable functions
  + Source code files called modules
  + Importing modules
  + Programs or scripts
  + Python execution model
* Modules
  + All python source file has extension .py
  + To run file type ‘python [filename].py’
  + To import into REPL
    - Type ‘python’ to enter REPL
    - Type ‘import [filename’
* Functions
  + Use def [function name](argurment list): to define a function
  + Ex)
    - def square(x):
    - return x \* x
  + \*\*: exponential operator
    - Ex) 2 \*\* 3 = 8
  + Naming special functions
    - \_ \_ feature \_ \_
  + Dunder
    - Our way of pronouncing special names
    - A portmanteau of “double underscore”
  + We’ll say “dunder name”
* \_\_name\_\_
  + Specially name variable allowing us to detect whether a module is run as a script or imported into another module
  + If you add print(\_\_name\_\_) to end of module, when you first import module it will print out file name
  + If you run module as a script using ‘python words.py’
    - It will print \_\_main\_\_
  + Python sets \_\_name\_\_ differently depending on how module is being used
  + Module can use this to determine how to behave
  + Ex)
    - if \_\_name\_\_ == ‘\_\_main\_\_’:
    - fetch\_words()
  + If \_\_name\_\_ is ‘\_\_main\_\_’ it is being executed, if it is not it is being loaded into another module
* The Python Execution Model
  + def is a statement
  + top level functions are defined when a module is imported or run
  + Python module
    - Convenient import with API
  + Python script
    - Convenient execution from the command line
  + Python program
    - Perhaps composed of many modules
* Command Line Arguments
  + Can import man multiple object from module using comma separated list
  + Ex)
    - from words import (fetch\_words, prints\_words)
  + ex) import everything from module
    - from words import \*
  + too access command line input
    - import sys
    - sys.argv[1]
  + sys.argv is a list of string, command line input
* Moment of Zen
  + Sparse is better than dense
  + Two between functions, number of lines PEP8 recommends
  + Single line for logical breaks
* Docstrings
  + Literal strings which document functions, modules and classes
  + Must be the first statement in the blocks for these constructs
  + Use “””
  + Ex)
    - “””Fetch a list of words from a URL.”””
  + Use help(function) to fetch docstring
  + help(words) to get all docstrings from module
* Comments
  + Docstrings explain how to consume how to use facilities rather than how it works
  + Code is ideally clear enough without ancillary explanation
  + Sometimes you need to explain why your code is written as it is
  + Comments in python start with # and extend to the end of the line
* Shebang
  + Have first line of script to include special comment called a shebang
  + #!/usr/bin/env python
  + Identify which interpreter to use
  + Document python version used
  + Locate python using PATH
  + For mac and linux, make script executable
    - chmod +x words.py
  + windows uses pylauncher
* Summary
  + Python code is generally placed in \*.py files
  + Executed modules by passing them as the first argument to pyhton
  + All top-level statements are executed when a module is imported
  + Define functions with the def keyword
  + Return objects from functions with the return keyword
  + Return without an argument returns None, as does the implicit return
  + Use \_\_name\_\_ to determine how a module is being used
    - If it is \_\_main\_\_ it is being executed as a program
  + A module is executed once, on first import
  + Def is a statement which binds code to a name
  + sys.argv contains command line arguments
    - argv[0] is the script file name
    - argv[1] is the first true arguments
  + dynamic typing supports generic programming
  + functions can have docstrings
    - “”” “”””
    - Provide usage information
  + Help( can retrieve docstrings
  + Modules can have docstrings
  + Python comments start with #
  + Program loaders can use #! To determine which python to run
    - Shebang
* Overview
  + Integer objects are immutable
    - When you reassign a variable python creates a new int object and references to the new object
    - Python garbage collector will clean it up at some point
  + id()
    - returns a unique integer identifier for an object that is constant for the life of the object
  + is
    - test equality of identity
    - if two references refer to the same object
  + assignment operator only binds objects to names
    - it never copies object by value
  + python doesn’t have variables in the sense of boxes holding a value
  + python has named references to objects
  + value vs identity equality
    - value-equality and identity equality are fundamentally different concepts
  + comparison by value can be controlled programmatically
  + identify comparison is unalterably defined by the language
* Passing Arguments and Return…
  + No copy of object is made
  + Function arguments are transferred using pass-by-object-reference
  + References to objects are copied, not the objects themselves
* Function Arguments
  + Can provide default argument values
  + Ex)
    - def banner(message, border=’-‘):
    - ….
  + Arguments with default values must come after those without default values
  + Can multiply string, the original string is repeated
  + Non default strings are positional argument
    - Matched by position
  + Default strings are key word arguments
    - Matched by name
    - Means you can pass in any order if you specify the name
  + def is a statement executed at runtime
  + default arguments are evaluated once when def is executed
  + Immutable default values don’t cause problems
  + Mutable default values can cause confusing effects
  + Ex)
    - def add\_spam(menu=[]):
    - menu.append(“spam”)
    - add\_spam() -> [“spam”]
    - add\_spam() -> [“spam”, “spam”]
  + default list is created exactly once when def statement is executed
  + Always use immutable object for default values
* Python’s Type System
  + Dynamic and strong typed
  + Dynamic type: type of object reference isn’t resolved until program is running
  + Strong typed: no implicit conversion between types
    - Ex) “45” + 2 -> results in an error
  + Python will not generally perform implicit conversions between types
    - Exception is in if,while statements, will convert to bool
* Scopes
  + Type declarations are unnecessary in python
  + Names can be rebound as necessary to objects of any type
  + Name resolution to objects is managed by scopes and scoping rules
  + Scopes in python
    - Local: inside the current function
    - Enclosing: inside closing functions
    - Global: at the top level of the module
    - Built-in: In the special builtins module
  + LEGB(local, enclosing, global, built-in)
  + Scopes in python do not correspond to source code blocks
  + Use global keyword to rebind global names into a local namespace
  + Ex)
    - count = 0
    - def set\_count(c)
    - global count
    - count = c
* Moment of Zen
  + Special cases aren’t special enough to break the rules
  + We follow patterns not to kill complexity but to master it
* Everything is an Object
  + dir(module): returns list of module attributes
* Summary
  + Python uses named references to objects
  + Assignment attaches a name to an object
  + Assigning one name to another makes them both point at the same object
  + The garbage collector removes objects with no references
  + id() returns a unique integer ID for an object
  + is determines if two names refer to the same object
  + we can test for equivalence using ==
  + function arguments are passed by object reference
  + rebinding function arguments loses the original object reference
  + return passes back an object reference to the caller
  + function arguments may have a default value
  + default argument values are evaluated once, when the function is defined
  + python uses dynamic typing
  + python has strong typing
  + python names are looked up using the LEGB rules
  + global references can be read from local scope
  + use global to assign to global reference from a local scope
  + everything in python is an object
  + import and def bind names to objects
  + type() reports the type of an object
  + dir() introspects the attributes of an object
  + you can access the name of a function or module with \_\_name\_\_
  + docstrings can be accessed through \_\_doc\_\_
  + you can use len() to measure the length of a string
  + the repetition operator \* repeats a string an integral number of times
* Overview
  + Str, list, dict, tuple, range, set
  + Protocols that unite collections
* Tuples
  + Immutable sequences of arbitrary objects
  + Once created objects in them cannot be replaced or removed
  + New objects cannot be added
  + Use () to create a tuple
  + Ex)
    - T = (“Norway”, 4.952, 3)
  + Access elements using []
  + Can have nested tuples
    - Use [][] to get inner elements
  + Ex)
    - T = ((220 , 284), (111, 333))
    - T[0][1] -> 284
  + Single object in () will be regarded as an int not a tuple
  + Ex)
    - H = (391)
  + Use a trailing comma separator
  + Ex)
    - H = (391,)
  + Use empty () to create an empty tuple
  + () can be omitted
  + Ex)
    - P = 1, 1, 1, 4, 5, 10
  + Tuple unpacking
    - Destructuring operation that unpacks data structures into named references
  + Ex)
    - Lower, upper = minmax([83, 33, 84])
  + This can be used to swap value
  + Ex)
    - a, b = b, a
  + use tuple(object) to convert object into tuple
  + use ‘in’ to test for member ship
  + ex)
    - 5 in (3, 5, 6) -> true
  + Use ‘not in’ to test for non member ship
  + Ex)
    - 5 not in (3, 5, 1) -> false
* Strings
  + Strings are immutable
  + You can not modify them in place
  + Use str.join() to join a collection of strings with a separator
    - Concatenation with + results in temporaries which can add up
    - str.join() inserts a separator between a collection of strings
    - call join() on the separator string
  + ex)
    - colors = ‘;’.join(‘blue’, ‘green’, ‘red’)
    - results in colors = ‘blue;gren;red’
  + Moment of zen: The way may not be obvious at first
  + To concatenate invoke join on empty text
  + string.parition(separator)
    - splits the string into 3 parts
    - before the separator
    - separator
    - after the separator
    - returns a tuple
  + use ‘\_’ for part we don’t care about in return value
  + ex)
    - origin, \_, destination = “Seattle-Boston”.partition(‘-‘)
  + string formatting
    - string interpolation
    - use {} in conjunction with string.format()
  + ex)
    - “{0} north, {1} east”.format(59.7, 10.4)
  + If the arguments are used once and in order you can use empty {}
  + Ex)
    - “{} nort, {} east”.format(59.7, 10.4)
  + PEP 498 has literal string interpolation
    - Commonly called f-strings
    - Embed expressions inside literal strings using a minimal syntax
    - Prefix string with ‘f’
  + Ex)
    - f"one plus one is {1 + 1}”
* Ranges
  + Sequence representing an arithmetic progression of integers
  + Ex) range(5)
  + Value provided to range is exclusive
  + range(start, end, step)
    - start is inclusive
    - end is exclusive
    - step is increment
    - if start is omitted, it will default to zero
  + ex)
    - range(0, 10, 2)
    - produces list [0, 2, 4, 6, 8]
  + range() signature
    - range(stop)
    - range(start, stop)
    - range(start, stop, step)
    - range does not support keyword arguments
  + always prefer to use iteration over object themselves
  + enumerate
    - constructs an iterable of (index, value) tuples around another iterable object
  + ex)
    - t = [6, 312, 51231, 12354341]
    - for p in enumerate(t):
    - print(p)
    - Prints -> (0, 6), (1,312) …
  + Ex)
    - for i, v in enumerate(t):
    - print(f”i = {i}, v = {v}”))
* Lists
  + Negative indexes
    - Index from the end of sequences using negative numbers
    - The last element is at index - 1
    - Negative indexes are one based since -0 = 0
  + Slicing
    - Extended form of indexing for referring to a portion of a list or other sequence
    - Syntax: a\_list[start:stop]
    - Stop is exclusive
    - If stop is blank, it will go to the end of the list
    - If start is blank, it will start at the beginning of the lis
    - If both are blank, entire list is returned
  + Slicing can be used to copy a list
    - s = [1 , 2, 3,4]
    - r = s[:]
    - r is a new list of [1,2,3, 4]
  + only references are copied
    - all elements in s and r point to the same items
    - if you reassign elements then things are find
    - if you edit elements then it reflect in both
  + ex)
    - a = [[1,2], [3,4]]
    - b = a[:]
    - a[0] = [8, 9]
    - then a is [[8,9], [3,4]]
    - and b is [[1,2], [3,4]]
    - a[1].append(5)
    - then a is [[8,9], [3,4,5]]
    - b is [[8,9], [3,4,5]]
  + all of these perform a shallow copy
    - create a new list that containing the same object references as the source list
  + shallow vs deep copy
    - shallow copies duplicate as little as possible
    - a shallow copy of a collection is a copy of the collection structure, not the elements
    - with a shallow copy, two collections now share the individual elements
    - deep copies duplicate everything
    - a deep copy of a collection is two collections with all of the elements in the original collection duplicated
  + use copy for deep copy
  + another example to be careful of is list multiplication
    - it copies the references
  + ex)
    - s = [[-1,1]] \* 3
    - s is then [[-1,1], [-1,1],[-1,1]]
    - s[2].append(7)
    - s is then [[-1,1,7], [-1,1,7],[-1,1,7]]
  + list.index()
    - find the location of an object in a list
    - returns the index of the first list element which is equal to the argument
    - will return error is not found
  + list.count(item)
    - returns how many times item appears in list
  + use ‘in’ and ‘not in’ to test for memberships
  + ‘del’ to remove an element from a list by index
    - Syntax: del a\_list[index]
  + List.remove(value)
    - Removes element from list by value
    - Value error if value not found
  + List.insert(index, value)
    - Insert and item into a list
    - Accepts an item and the index of the new item
  + ‘ ‘.join(string list) to join list into a string
  + List.extend(list) to as list to existing list
  + List.reverse()
    - Modify a list in place
  + List.sort()
    - Modify a list in place
    - Accepts reverse parameter
    - Ex) d.sort(reverse=True) to sort in reverse
    - Accepts key parameter
  + Key parameter to list.sort()
    - Can be any callable object that accepts a single parameter
    - Items passed to callable and sorted on its return value
    - Ex) h.sort(key=len)
    - Sort by length of key
  + reversed()
    - out-of-place equivalents to list.reverse()
    - returns the new list in reverse
  + sorted()
    - out-of-place equivalent to list.sort()
    - returns list\_reverse iterator
    - use iterator to create a list
  + ex)
    - p = [9, 3, 1, 0]
    - q = reversed(p)
    - list(q)
* Dictionaries
  + Keys must be unique
  + Internally dictionary maintains pairs of references to key and value objects
  + Keys must by immutable
  + Values may be mutable
  + Do not rely on the order of items in the dictionary
  + dict([(‘Alice’, 32), (‘Bob’, 48)] to convert to dictionary
  + dictionary copying is shallow
    - copying only references not the object themselves
  + use copy() to copy dictionary
  + ex)
    - b = a.copy()
  + second is to use the dict() constructor and pass in an existing dictionary
    - b = dict(a)
  + dict.update()
    - adds entries from one dictionary into another
    - call this on the dictionary that is to be updated
    - if keys in the source dictionary is already present then values in the target dictionary are replaced from the source
  + dictionary iteration
    - dictionaries yield the next key on each iteration
    - value can be retrieved using the square-bracket operator
  + to iterate over values
  + ex)
    - for value in colors.values():
  + To iterate over keys
    - for value in colors.keys():
  + dict.items()
    - iterators over keys and values in tandem
    - yields (key, value) tuple on each iteration
  + use ‘in’ and ‘not in’ to check for key membership
  + del to delete from dictionary
  + ex)
    - del z[‘Fy’]
* Sets
  + Unordered collection of unique elements
  + sets are mutable
  + elements in a set must be immutable
  + {} is a set
    - ex) p = {6,28,1,4}
  + sets are unordered, items you added may be in a different order than input order
  + set(): constructor to create a set
  + d = {} creates an empty dictionary
  + sets cannot have duplicate items
  + can convert other collections to set to remove duplicates
  + is ‘in’ and ‘not in’ to test for membership
  + use set.add() to add to set
  + use set.update() to add another collection to the set
  + set.remove(item) to remove item from set
    - if item is not in set then it will return an KeyError
  + set.discard(item) to remove item from set
    - will not return error if item is not in the set
  + set.copy() to perform shallow copy of set
    - copies references but not objects
  + most useful aspect is set algebra
    - union, difference, intersection, subset, superset, disjoint
  + set.union(set2)
    - all elements in either or both sets
    - commutative
  + set.intersection(set2)
    - all elements present in both sets
    - commutative
  + set.difference(set2)
    - elements present in set 1 but not in set 2
    - not commutative
  + set.symmertic\_difference(set2)
    - elements in set 1 and set 2 but not in both
    - commutative
  + set.issubset(set2)
    - all elements in set 1 is in set 2
  + set.issuperset(set2)
    - all elements in set 2 is in set 1
  + set.isdisjoint(set2)
    - checks that two sets have no members in common
* Protocols
  + a set of operations that a type must support to implement the protocol
  + do not need to be defined as interfaces or base classes
  + types only need to provide functioning implementations
* Summary
  + tuples are immutable sequence
  + tuples literals are optional parentheses around comma-separated items
  + using a trailing comma for single-element tuples
  + tuple unpacking is useful for multiple return values and swapping
  + use str.join() for efficient string concatenation
  + use str.partition() for certain simple string parsing operations
  + str.format() is a powerful string interpolation technique
  + f-strings are a kind of string literal for performing interpolation
  + range objects represent arithmetic progressions of integers
  + range() can be called with one, two, or three arguments
    - start, stop, step
  + enumerate is often better than range for making loop counters
  + lists supports indexing from the end with negative indices
  + slicing copies all or part of a list
  + the full slice is a common idiom for copying lists
  + use list.index() and list.count() to look for elements in a list
  + use ‘del’ keyword to remove elements from a list
  + sort and reverse lists in-place with list.sort() and list.reverse()
  + sorted() and reversed() sort and reverse any iterable
  + Lists copies are shallow, only copying the references
  + dictionaries map from keys to values
  + iteration and membership in dictionaries are over keys
  + do not assume any order when iterating dictionary keys
  + dict.keys(), dict.values(), and dict.items() are iterable views into dictionaries
  + copy dictionaries with dict.copy() or the dict constructor
  + use dict.update() to extend one dictionary with another
  + sets are unordered collections of unique elements
  + sets support powerful set-algebra operations and predicates
  + built-in collections can be organized by protocols
  + underscore often represents unused values
  + the pprint module support pretty printing of compound data structures
* Overview
  + Exception concepts
  + raising exceptions
  + control flow
  + catching exceptions
  + unhandled exceptions
  + Exception handling
    - mechanism for interrupting normal program flow and continuing in surrounding context
  + Exceptions: Key Concepts
    - raising an exception
    - handling an exception
    - unhandled exceptions
    - exception objects
* Exceptions and Control Flow
  + exception return is in the following type “exceptionType: payload”
  + Exception propagates across several levels
* Handling Exceptions
  + use try-except construct
  + ex)
    - def convert(s):
    - try: #raise exceptions
    - ….
    - except KeyError: #handle exceptions
    - x = -1
    - …
  + can have multiple except blocks to catch different type of errors
  + ex)
    - try:
    - ….
    - except KeyError:
    - …
    - except TypeError:
    - ….
  + can also merge exception blocks
  + ex)
    - try:
    - ….
    - except(KeyError, TypeError):
    - ….
* Exceptions and Programmer Er…
  + empty blocks are not permitted
  + exceptions resulting from programmer errors
    - IndentationError
    - SyntaxError
    - NameError
    - should not be handled with exception handling, should be fixed during development rather than runtime
  + ‘pass’ keyword, is a no-op
    - it does nothing
    - purpose is have “empty” blocks
  + accessing Exception Objects
  + use ‘as’ keyword
  + ex)
    - try:
    - except (KeyError, TypeError) as e:
    - …
  + f”{expr!r}
    - including !r after expression the repl representation value will be inputted into the string
* Re-raising Exceptions
  + Exceptions can not be ignored
  + Error codes are easy to ignore
  + Checks are always required
  + Re raise an exception using ‘raise’ keyword
  + Ex)
    - try:
    - …
    - except (KeyError, TypeError) as e:
    - print(“..”)
    - raise
  + ‘raise’
    - Without parameter, raise simply reraises the exception currently being handled
* Exceptions Are Part of the API
  + ‘and’ tests both conditions are true
  + Ex)
    - while guess \* guess != x and i < 20:
    - ..
  + ‘or’ test either conditions are true
  + Standard Exception Types
    - Python provides standard exceptions types for signaling common errors
  + Invalid argument values
    - Use ValueError for arguments of the right type but with an invalid value
  + Exception Constructors
    - Use ‘raise’ ValueError() to raise a new ValueError
* Exceptions and Protocols
  + Sequences should raise IndexError for out-of-bounds indexing
  + Exceptions must be implemented and documented correctly
  + Existing built-in exceptions are often the right one to use
  + Follow existing patterns
    - The more your code follows established patterns, the easier it will be for others to use
  + IndexError
    - An integer index is out of range
  + ValueError
    - Object is of the correct type but has an inappropriate value
  + KeyError
    - A lookup in a mapping failed
* Avoid Explicit Type Checks
  + Avoid catching TypeError
    - Increase function reusability
    - Let TypeError arise on their own
* It’s Easier to Ask Forgiveness T..
  + Check all preconditions
  + Prepare for consequences
  + LBYL: look before you leap
  + EAFP: easier to ask forgiveness than permission
  + Python prefers EAFP
    - Code’s happy path is emphasized rather than being interspersed with error handling
  + Process file LBYL approach
  + Ex)
    - import os
    - p = ‘/path/to/datafile.dat’
    - if os.path.exists(p): #checks if the files exists before use
    - process\_file(p)
    - else
    - print(f’No such file as {p}’)
  + Process file EAFP approach
  + Ex)
    - p = ‘/path/to/datafile.dat’
    - try:
    - process\_file(f)
    - except OSError as e: #handle error
    - print(f’Error: {e}’)
  + EAFP is enabled by exceptiosn
  + Without exceptions, error handling is interspersed in program flow
  + Exceptions can be handled non-locally
  + EAFP plus Exceptions
    - Exceptions are not easily ignored
    - Error codes are silent by default
    - Exceptions plus EAFP makes it hard for problems to be silently ignored
* Cleanup Actions
  + Use try-finally
  + Ex)
    - try:
    - ….
    - finally:
    - ….. #executed no matter how the try-block terminates
  + The finally-block always executes
  + Moment of Zen
    - Errors should never pass silently, unless explicitly silenced
    - Errors are like bells, if we make them silent then they are of no use
* Platform-Specific Code
  + For windows use ‘msvcrt’ module
  + For linux and mac use ‘tty’, ‘termios’ and ‘sys’ modules
  + Ex)
    - try:
    - import msvcrt
    - def getkey():
    - return msvcrt.getch()
    - except ImportError:
    - import sys
    - import tty
    - import termios
    - def getKey():
    - fd = sys.stdin.fileno()
  + The caller can take alternative actions if both imports fial
  + For example, they could downgrade to using input()
* Summary
  + Raising an exception interrupts program flow
  + Handle exceptions with try…except
  + Exceptions can be detected within try-blocks
  + Except-blocks define handlers for exceptions
  + Python uses exceptions pervasively
  + Except-blocks can capture the exception
  + Avoid catching programmer errors
  + Signal exceptional conditions with ‘raise’
  + ‘raise’ without an argument re-raises the current exception
  + Generally don’t catch TypeError
    - Otherwise would negate dynamic typing
  + Use str() to convert exceptions to strings
  + Exceptions are part of an API
  + Prefer built-in exception types when possible
  + Use try..finally for cleanup
  + Print() output can be redirected
  + !r forces REPR representations in f-strings
  + Python support logical ‘and’ and ‘or’ operators
  + Return codes are too easily ignored
  + Implement platform-specific actions with ImportError and EAFP
* Overview
  + Comprehensions
    - Creating familiar objects
    - Creating new kinds of objects
    - Filtering
  + Low-level iterable API
    - Iterators
    - Exceptions in interation
  + Generator functions
    - The ‘yield’ keyword
    - Statefulness, laziness, and infinite sequences
    - Generator expression
  + Iteration tools
* List and Set Comprehensions
  + Concise syntax for describing lists, sets, and dictionaries
  + Readable and expressive
  + Close to natural language
  + List Comprehensions uses []
  + Ex)
    - [len(word) for word in words]
  + Ex w/o comprehensions
    - lengths = []
    - for word in words:
    - lengths.append(len(word))
  + Evaluate and create new value to become new value in list
  + Syntax
    - [expr(item) for item in iterable]
  + The expression production the new list’s elements can be any python expression
  + Set comprehensions uses {}
  + Ex)
    - s = { len(str(factorial(x))) for x in range(20)}
* Dictionary Comprehensions
  + Syntax
    - {
    - key\_expr(item): value\_expr(item)
    - for item in iterable
    - }
  + Ex)
    - capital\_to\_country = { capital: country for country, capital in country\_to\_capital.items() }
  + dictionary comprehensions don’t work directly on dict sources
  + use dict.items() to get keys and values from dict sources
  + ex)
    - { x[0]: x for x in words }
  + Comprehensions expression can be arbitrarily complex
  + Avoid excessive complexity
  + Put complex expression in separate functions for readability
* Filtering Comprehensions
  + Filter clause
  + Create a function to return true of false based on filtering condition
  + Then use it in comprehension
  + Ex)
    - [x in x in range(101) if is\_prime(x)]
* Moment of Zen
  + Simple is better than complex
  + Code is written once but read over and over
  + Fewer is clearer
  + Comprehensions should normally have no side-effects
* Iteration Protocols
  + Comprehensions and for loop iterate of whole sequence
  + Iterable
    - Can be passed to iter() to produce an iterator
  + Iterator
    - Can be passed to next() to get the next value in the sequence
  + Ex)
    - iterable = [‘Spring’, ‘Summer’, ‘Autumn’, ‘Winter’]
    - iterator = iter(iterable)
    - next(iterator) #’Spring’
    - next(iterator) #’Summer’
  + if you call next(iterator) on the list element in list, it will return an StopIteration exception
  + stopping iteration with an exception
    - sequences only have one ending, so reaching it is exceptional
    - finding the end of an infinite sequence would by truly exceptional
* Generator Functions
  + Iterables defined by functions
  + Lazy evaluation
    - Only compute next value on demand
  + Can model sequences with no definite end
  + Generator functions must include at least one yield statement
  + They may also include return statements
  + Ex)
    - def gen123():
    - yield 1
    - yield 2
    - yield 3
  + Assigning gen123() to a variable, variable references a generator object
    - Ex) g = gen123()
  + Generator object is python iterators
  + Call next(g) returns the yield value in order
  + Ex)
    - next(g) #1
    - next(g) #2
    - next(g) #3
  + each call to generator functions returns a new generator object
* Maintaining State in Generators
  + Use graphical debugger
  + Pycharm is good, can use any
  + ‘continue’ finish current loop iteration and being the next iteration immediately
  + Lazy computation can result in complex flow control
  + Forced evaluation can simplify things during development
* Laziness and the Infinite
  + Generators only do enough work to produce requested data
  + This allows generators to model infinite(or just very large sequences)
* Generator Expressions
  + Cross of comprehension and generator function
  + Similar syntax to comprehension
  + Result in create of generator object
  + Syntax:
    - (expr(item) for item in iterable)
  + Ex)
    - Million\_squares = (x\*x for x in range(1, 1000001))
  + Generator are single use objects
  + To recreate a generator from a generator expression, you must execute the expression again
* Iteration Tools
  + Python provides a powerful vocabulary for working with iterators
  + These include the familiar enumerate() and sum()
  + The itertools module provides many more
  + Itertools.islice()
    - Perform lazy slicing of any iterator
  + Ex)
    - from intertools import islice
    - islice(all\_primes, 1000)
  + itertools.count()
    - an unbounded arithmetic sequence of integers
  + Boolean aggregation
    - any(): determines if any elements in a series are true
    - all(): determines if all elements in a series are true
  + ex)
    - any([False, False, True]) #True
    - all([False, False, True]) #False
  + zip()
    - synchronize iteration across two or more iterables
  + ex)
    - sunday = [ 12, 14, 15, ..]
    - monday = [13,14,14,…]
    - for item in zip(sunday, monday):
    - print(item)
  + Out put would be (12, 13), (14,14), (15,14) ….
  + intertools.chain()
  + ex)
    - from intertools import chain
    - temperatures = chain(sunday, monday, tuesday)
    - all(t > 0 for t in temperatures) #True
* Summary
  + Comprehensions for list, set, and dict
  + Comprehension use an input iterable and an optional predicate
  + Iterable objects can be iterated item by item
  + Use iter() to get an iterator from an iterable object
  + Use next() to get the next item from an iterable
  + Iterators raise StopIteration when they’re exhausted
  + Generator functions describe sequences imperatively
  + Generator functions contain at least one yield
  + Generators are iterators
  + Each call to a generator function produces a new generator
  + Generators maintain explicit internal state
  + Generators yield values lazily
  + Generator expressions are a type of comprehension that creates generators
  + Built-in iteration tools include sum(), any(), and zip()
  + The itertools module includes many other tools for iteration
* Overview
  + Instance methods
  + Self arguments
  + Initializers
  + Constructors
  + Invariants
  + Collaborating classes
  + Decomposing problems
  + Interface and implementation
  + Combine programming paradigms
  + Everything is an object
  + Nominal typing and duck typing
  + Inheritance
* Classes
  + Classes define the structure and behavior of objects
  + Acts as a template for creating new objects
  + Classes control an object’s initial state, attributes, and methods
  + Classes can make complex problems tractable
  + But they can make simple problems unnecessarily complex
  + Python lets you strike the right balance between functions and classes
* Defining Classes
  + class MyClassName:
  + by convention, class names uses CamelCase
  + ex)
    - class Flight:
    - pass
    - f = Flight()
  + type(f) #class Flight
* Instance Methods
  + Functions defined within class
  + Method must accept a reference to actually instance to which method was called
    - Called ‘self’
  + Ex)
    - class Flight:
    - def number(self):
    - return “SN060”
  + f.number() is syntactic sugar for Flight.number(f)
* Instance Initializers
  + \_\_init\_\_()
    - Instance method for initializing new objects
    - Must also accept ‘self’
    - Is an initializer, not a constructor
    - Configure an object that already exists
  + Ex)
    - \_\_init\_\_(self, number):
    - self.\_number = number
  + Self is similar to this in java, c#, etc
  + Assigning to object attribute that does not exists is enough to bring it in existence
  + Why \_number?
    - Avoid name clash with number()
    - By convention implementation details start with underscore
  + Everything is public
    - No private, public, protected
  + Generally do not directly access attributes
  + Class invariants
    - Truths about an object that endure for its lifetime
  + Ex)
    - def \_\_init\_\_(self.number):
    - if not number[:2].isalpha():
    - raise ValueError(f”No airline code in ‘{number}’”)
    - self.\_number = number
  + ‘not’ logical negation operator
* A Second Class
* Collaborating Classes
  + The Law of Demeter
    - The principle of least knowledge
    - Only talk to your friends
    - Object oriented design principle
    - Never call method on objects you received from other calls
* Moment of Zen
  + Complex is better than complicated
  + Many moving parts combined in a clever box are now one good tool
* Booking Seats
* Methods for Implementation D…
* Object-Oriented Design with F..
  + Separation of concerns
  + Functions are objects too
  + Tell! Don’t ask.
    - Tell other objects what to do instead of asking them their state and responding to it
  + ‘\’: line continuation
    - Allow us to split a long statement over several lines
  + Ex)
    - output = f”| Name: {passenger}” \
    - f” Flight: {flight\_number}” \
    - f” Seat: {seat}” \
    - f" Aircraft: {aircraft}” \
    - “ |”
* Polymorphism and Duck Typing
  + Using objects of different types through a uniform interface
  + It applies to both functions as well as more complex types
  + make\_boarding\_card() did not rely on any concrete types
    - any other object that fit the interface would work in place of console\_card\_printer()
  + polymorphism is achieved through duck typing
    - “When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.”
  + An object’s fitness for use is only determined at use
  + This is in contrast to statically typed compiled languages
  + Suitability is not determined by inheritance or interfaces
* Inheritance and Implementatio…
  + Inheritance: mechanism where a class(sub class) can be derived from another class(base class)
    - Allows us to make behavior more specific in the sub class
  + Nominally-typed languages use inheritance for polymorphism
  + Python uses late binding
    - Late binding
    - No python method calls or attribute lookup are bound to actually objects until they are called
  + You can try and method on any object
    - Attempt polymorphism on an object and see if it fits
  + Inheritance in python is primarily useful for sharing implementation between classes
  + Inheritance can be used to facilitate polymorphism
  + Inheritance syntax
    - class Sub\_Class(Base\_Clase):
  + thanks to duck-typing python uses inheritance less than many other languages
* Summary
  + All types in python have a class
  + Classes define the structure and behavior of objects
  + An object’s class is set when it’s created, and it’s fixed for the object’s lifetime
  + Classes are a key part of object-oriented programming in python
  + Classes are defined with the class keyword
  + Instances of a class are created by calling the class like a function
  + Instance methods are functions defined within a class and must accept a self argument
  + Methods are called using the instance.method() syntax
  + Classes may have a \_\_init\_\_() method for initializing new instances
  + A class’s constructor will call \_\_init\_\_() if it’s present
  + \_\_init\_\_() is not, strictly speaking a constructor
  + Arguments passed to the constructor are forwarded to \_\_init\_\_()
  + Instance attributes are created simply by assigning to them
  + Implementation details are conventionally prefixed with an underscore
  + Access to implementation details outside a class can be useful during development
  + Class invariants should be established within \_\_init\_\_()
  + Methods can have docstrings “”””
  + Classes can have docstrings
  + Method calls on self within a method must be preceded with self
  + A module can contain as many classes and functions as you wish
  + Polymorphism in python is achieved through duck-typing
  + Polymorphism in python doesn’t rely on shared base classes or interfaces
  + Class inheritance in python is primarily useful for sharing implementation
  + All methods - including special methods - are inherited
  + Strings support slicing
  + The Law of Demeter can help reduce coupling
  + We can nest comprehensions
  + It can be useful to discard the item in a comprehension, conventionally using underscore
  + You can discard a collection element to simplify handling one-based indexing
  + Don’t feel compelled to use classes if functions suffice
  + You can spread complex comprehensions over multiple lines
  + Statements can be split over lines with backslash; use this primarily to improve readability
  + The “Tell! Don’t ask.” Approach to object-oriented design can reduce coupling
* Overview
  + Resources
    - Program elements that must be released or closed after use
    - Python provides special syntax for managing resources
  + Text vs binary mode
  + Context managers
* Opening Files
  + open()
    - open a file for reading or writing
    - file: the path to the file(required)
    - mode: read, write, or append, plus binary or text
    - encoding: encoding to use in text mode
  + files are stored as binary
  + binary mode
    - write() and read() as byte objects without decoding
  + Text mode
    - Encode() -> write()
    - Decode() read()
    - Universal newlines
  + If you don’t specify encoding, python will use default ‘utf-8’, not always guaranteed
* Writing Text
  + Ex)
    - f = open(‘wasteland.txt’, mode=’wt’, encoding=’utf-8’(
    - w is for writing
    - t is text
  + open() Modes
    - ‘r’: open for reading
    - ‘w’: open for writing
    - ‘a’: open for appending
  + Selector
    - ‘b’: binary mode
    - ‘t’: text mode
  + ‘wb’: open for writing in binary mode
  + ‘at’: open for appending in text mode
  + open() returns a file-like object
  + help() works on modules, methods, and types and it works on instances too
  + after to finish call close on file
  + ex)
    - f.close()
  + write() returns the number of codepoints written
  + don’t sum these values to determine file length
* Reading Text
  + Mode: rt
  + Use read(num char)
    - Num char is number of characters to read
    - After read, advanced file pointer to past what was read
  + read()
    - without any parameters will read the rest of the file
  + seek(location)
    - return file pointer to location
    - 0 for start of file
  + Seek cannot move to arbitrary offset
  + Only 0 and values from tell() are allowed
  + Other values result in undefined behavior
  + readline()
    - read till the next newline character
    - or end of file if no newline character exist
  + readlines()
    - read all lines into list
    - separate by newline
* Appending Text
  + writelines() which writes iterable series of string to stream
  + must provide newline yourself
* File Iteration
  + Ex)
    - f = open(sys.argv[1], mode=’rt’, encoding=’utf-8’)
    - for line in f:
    - print(line)
    - f.close()
  + use sys.stdout.write() instead of print
    - it won’t add newline like print()
* Closing Files with Finally
  + Ex)
    - try:
    - f = open(…..)
    - …..
    - finally:
    - f.close()
* With-blocks
  + File usage pattern
    - f = open(…)
    - #work with file
    - f.close()
  + if you don’t close, you can lose data
  + we want a mechanism to pair open() and close() automatically
  + with-block
    - control flow structure for managing resources
    - can be used with any objects - such as files - which support the context-manager protocol
  + ex)
    - def read\_series(filename):
    - with open(filename, mode=’rt’, encoding=’utf-8’) as f:
    - return [int(line.strip()) for line in f]
* Moment of Zen
  + Beautiful is better than ugly
  + Sugary syntax
  + Faultlessness attained through sweet fidelity
  + Expansion of the with-block
  + Ex)
    - with EXPR as VAR:
    - BLOCK
* Binary Files
* Bitwise Operators
* Pixel Data
* Reading Binary Data
* File-like objects
  + Objects that behave like files
  + A semi-formal protocol
  + File behaviors are too varied for a fully-specified protocol
  + Use an EAFP approach with file-like objects when necessary
* Context Managers
  + from contextlib import closing
  + ex)
    - …
    - def raid(food):
    - with closing(RefrigeratorRaider()) as r:
    - …..
  + with closing will call close() for us
* Summary
  + Files are opened with open()
  + Specify encoding for text-mode files
  + Text-mode works with str and does universal newline translation and encoding
  + Binary-mode works with bytes objects
  + Callers are responsible for providing newlines
  + Files should always be closed after use
  + Files have line-reading support and yield lines on iteration
  + Files are context managers which close files on exit
  + The notion of file-like object is loosely defined by very useful
  + Context managers aren’t restricted to file-like objects
  + Help() can be called on objects, not just types
  + Python supports the bitwise operators &, |, << and >>