Quick Python Intro

Python Tutorial

Outline

- Basics
- Control Flow
- File I/O
- Classes
- Thoughts on Efficiency
- Practice Problems

Python Basics

print("Hello, Python!")

- Python is interpreted
- parts may get compiled (.pyc files show up)
- Running code: python3 or py command.
  - Run a file as a script: demo$ python3 somefile.py
  - Load interactive mode: demo$ python3 –i
  - Load a file interactively: demo$ python3 –i somefile.py
- Suggestion: keep reloading your script, explore the next step, update your file with the good stuff.

Basic Types

- **int**: unbounded integers. (Yes, they are objects! Can't call functions on them though)
- **float**: 64-bit double-precision floating point numbers.
- **bool**: True or False. (capitalized)
- **string**: sequence of unicode characters.
  - Can use single, double, or triple quotes (triple allows newlines within)
  - b"stuff" is a byte string. Avoid unless you're really playing with space/layout
  - f"insert {var} values inline!"

A few operations

- Math: + – * / // (int div) ** (exponent)
- Booleans: and or not
- Relational operators: < <= > >=
  - Can be chained: w < x <= y > z (is w<z? not checked…)
    (w<x) and (x<=y) and (y>z)
  - Short-circuited: 2>4>x quits as soon as 2>4 is False.

Compound Types

- **list**: sequence of any python values. (array-list impl.)
  - some operations: .append(), .extend(), .pop(), .insert(), .sort() …
- **tuple**: immutable version of lists.
- **dict**: dictionary of key-value pairs. (hash table implementation)
  - Keys must be hashable ("immutable all the way down" will suffice)
  - Some operations: len(), .get(), del, .pop()/.popitem(), .copy(), …
- **set**: mutable unordered group of values. (vals must be hashable)
  - Some operations: |, &, ^, -,

Indexing things

- Zero-based indexing going forward
- Negative-one-based indexing going backward
- IndexError thrown when out of bounds

msg = "index"

0	1	2	3	4
i	n	d	e	x
-5	-4	-3	-2	-1

xs = [8.5, 100, -16.3, 2.5]

0	1	2	3
8.5	100	-16.3	2.5
-4	-3	-2	-1

Sequence Operations

operation	meaning	result type
x in s	checks if an item in s equals x.	bool
x not in s	checks if no items in s equal x.	bool
s + t	concatenation	same seq. type
s*n (or: n*s)	n shallow copies of s, concatenated	same seq. type
len(s)	length of s	int
s.count(x)	find # items in s equal to x	int (#matches)
s.index(x[,i[,j]])	give index of first x in s. (if not found, crashes)	int

(these are all expressions)

Strings

Some String Methods (See LIB 4.7.1)

usage: stringExpr . methodName (args)

method	purpose	returned value
`s.upper()` `s.lower()`	converts letters to upper or lower case	modified copy of s
`s.startswith(svar[,start[,stop]])` `s.endswith(svar[,start[,stop]])`	is svar a prefix/suffix of s?	bool
`s.join(iterable)`	concatenates items from iterable, with copies of string s inbetween them.	string result of all those joined things
`s.split(sep)`	get list of strings obtained by splitting s into parts at each occurrence of sep.	list of strings from between occurrences of sep
`s.replace(old, new[,count])`	replace all (or count) occurrences of old str with new str.	string with replacements performed

Formatting Strings

Brief introductions here, but also read the documentation.

Three approaches:

- percent operator, **%** *LIB 4.7.2 ZY 3.7*
- **format** method *LIB 6.1.3 ZY 7.5*
- f-strings *new as of Python 3.6*

String Formatting: % operator

- describe pattern of string with placeholders, then supply all substitutions at once.
- Syntax: `pattern_string % tuple`
- Semantics:
  → simplify lefthand string to value
  → left to right, match placeholders in string with values from tuple
  → substitutions obey special formatting directives

String Formatting: % operator

placeholder	style of output	accepted input
%d	integer	integers, floats
%f	float	integers, floats
%g	float (scientific notation)	integers, floats – but it prefers scientific notation representation
%s	string	anything (calls str())
%%	the '%' character	none – just represents the % symbol
<more>	…	don't memorize these: %i, %o, %u, %x, %X, %e, %E, %c, %r…

```
"there are %d days until %s."  %  (75, "holiday")
"%s ran %f miles today"%  ("Zeke", 3.5)
"change is %d dollars and %d cents."  %  (4,39)
"you got %f%% of them, %s."  % (0.95*100, "George")
"%s's number is %g/mol."  %  ("Avogadro",6.02214e23)
```

More Options

purpose	examples	results
state exact # columns after decimal point (%f)	`"%.2f" % (2/3)` `"%.0f" % 15.5`	`'0.67'` `'16'`

state min. # columns for entire thing	`"%4d"   %   30`
	`"%3d"   %   1234`
	`"%5f"   %   2.5`
		`'  30'`
		`'1234'`
		`'  2.5'`

use leading sign (+/-)	`"%+f"   %   5`	`'+5.000000'`

The **format( )** method

A powerful option to craft a string is the format method.

- use { }'s as placeholders, put style rules inside
  - examples: `"{}"` `"{:4.2f}"` `"{:^5}"`
- provide substitutions as arguments to .format() method

See more examples: LIB 6.1.3.2

More options…

Again, indicate min # cols and exact # cols after the decimal.

```
"{:10.2}".format(0.123)
```

```
'      0.12'
```

- Show as percent:

```
"{:%}".format(0.12)
```

```
'12.000000%'
```

- Align left/center/right:

```
"{:>6}".format("hi")
"{:<6}".format("hi")
"{:^6}".format("hi")
"{:.^6}".format("hi")
"{:o^6}".format("hi")
```

```
'    hi'
'hi    '
'  hi  '
'..hi..'
'oohioo'
```

New! f-Strings

- special version of string literals with embedded expressions
- indicated with leading f in front of open-quote.

```
name = "George"
age = 67
print( f"Happy {age}th birthday, {name}!")
```

Some corner cases of f-strings

- you can call functions and methods:
  - f"{len(name)} letters long"
  - f"{name.upper()}!"
- watch out for clashing quote styles!
  (can't \escape them inside the {}'s)
  - f"{1+3} years"
  - f"{int('1')+3} years"
  - f"{int(\"1\")+3} years"   *FORBIDDEN*

printing

- Feed any number of args to print()
- Keyword arguments of note:
  - sep: separator between printed items (default: sep=" ")
  - end: thing printed once at the end of printing (default: end="\n")
  - flush: require printing partial line now? (vs. waiting for a newline)

```
>>> print(2,4,6)          #sep=" ", end="\n"
2 4 6
>>> print("hi", 2, True, sep=",", end="!!!")
hi,2,True!!!>>>
```

lists

lists: mutable sequences

- When a sequence is mutable (as lists are), we can update part of the structure, leaving the rest alone:

```
xs = [1,2,3]
xs[1] = 99
print(xs)          # prints out  [1, 99, 3]
```

- There are many operations available on mutable sequences (see next slides).

list update/delete operations

operation	meaning
s[i] = x	replace ith item of s with x
s[i:j] = t	replace slice i:j with t. lengths needn't match!)
s[i:j:k] = t	replace slice i:j:k with t. (lengths *must* match!)
del s[i]	remove ith item from s.
del s[i:j]	remove slice i:j from s.
del s[i:j:k]	remove slice i:j:k from s.

try interactively.

list operations

operation	meaning	returned value
s.append(x)	add x as a single value at end of s.	None value
s.extend(t)	individually append each item of t to the end of s.	None value
s.insert(i,x)	make space (push other spots to the right), put x value at location i.	None value
s.pop(i)	remove value at index i from sequence; return the value that was there	item that was at index i
s.remove(x)	find first occurrence of x, remove it.	None
s.reverse()	reverse the ordering of items.	None
s.sort()	sort the items in increasing order.	None

append: attach a value to the list.
extend: attach a sequence to the list.

try interactively.

Programming **TRAP**

- many mutable sequence operations return the **None** value

→ value is directly modified: rather than returning a modified copy, returns the **None** value

→ assigning the result back to the variable discards the value!

```
xs = [2,5,4,1,3]
ys = [2,5,4,1,3]
xs.sort()
ys = ys.sort()
print (xs, type(xs))
print (ys, type(ys))
```

output when run:

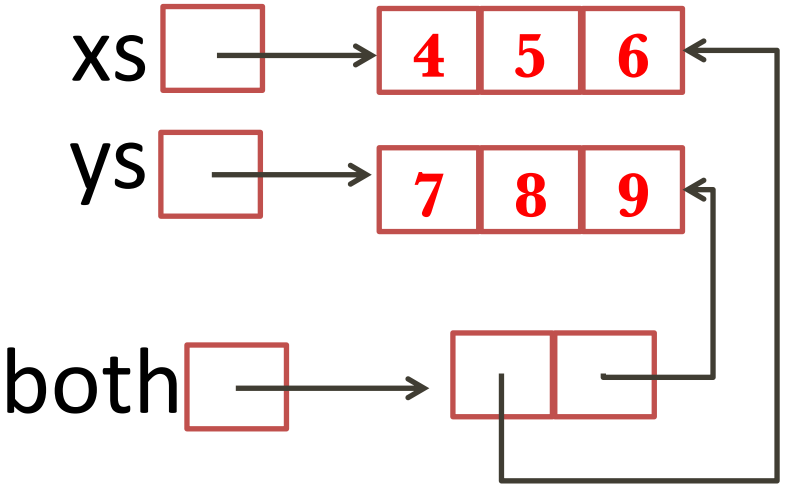
```
[1, 2, 3, 4, 5] <class 'list'>
None <class 'NoneType'>
```

ys did get sorted, but then we threw out the whole list by storing a ***None*** *value into ys.*

Memory Usage

- These arrows help us understand complex data, such as lists of lists.
- Every **variable** always stores one value in a box.
- The only new concept is that sometimes the contents of the box is an arrow (a reference) to some other value in memory.

```
xs = [4,5,6]
ys = [7,8,9]
both = [xs,ys]
```



List ideas

- Start empty, append as you go. Feasible in multiple dimensions.

```
grid = []
for row_i in range(3):
    row = []
    for col_i in range(5):
        item = (row_i, col_i)
        row.append(item)
    grid.append(row)
```

Control Flow

Statements, assignment

- Sequential lines are run in order.
- function definitions need to be executed before actual calls
  - code in function not inspected until it's called, other than syntax
  - thus functions can be defined in any order and called later (e.g. main)

- Statements:
  - The usual assignment: location = expression
  - Increment: x = x + 1 x += 1 (can't use x++)

Control Flow Statements

- Selection: if, if-else, if-elif*, if-elif-else
- Loops: while, for
  - break and continue are available;
- function calls, recursion

Branching Examples

- All styles of branching/loops can be nested; just indent further.
  - *Don't mix tabs/spaces!*
- Use elif, not nested else: if:
- No switch/case statement.

```
if expr:
   stmts
```

```
if expr:
   stmts1
else:
   stmts2
```

```
if expr1:
   stmts1
elif expr2:
   stmts2
elif expr3:
   stmts3
…
else:
   stmtsN
```

while statement

- expr: boolean expression
  - if True, run loop body and try again.
  - Note: non-bool things will be interpreted as bools! Non-zero/non-empty means True, zero/empty means false… bad habit.

```
while expr:
    stmts
```

for statement

- General form:

```
for newvar in sequenceExpr :
    stmts
```

- common form:

```
xs = ... # some sequence
for i in range( len(xs) ):
    ...xs[i]...
```

"Value" For-Loop (foreach loop)

- For-loops assign each value of the supplied sequence to the loop variable.
- We directly traverse the values in the list themselves

```
# sum up some numbers.
vals = [1.5, 2.25, 10.75, -2.0]
total = 0
for curr_val in vals:
    total += curr_val
print("sum of vals is",total)
```

```
# what is the largest value?
vals = [17, 10, 99, 14, 50]
max_val = vals[0]
for val in vals:
    if val > max_val:
        max_val = val
print("largest:",max_val)
```

```
# print some words out.
words = ["you", "are", "great"]
for word in words:
    print(word)
```

"Index" For-Loop (traditional-ish for-loop)

We can generate all the valid *indexes* we'd like to visit, and supply those to a for-loop instead of the values-sequence itself.

We are thus aware of our position (i) as well as the value at the current position (vals[i])

```
# where is the largest value located?
vals = [2,5,3,6,4,1]
max_loc = 0
for i in range(len(vals)):
    if vals[i]>vals[max_loc]:
        max_loc = i
print("maxval="+str(vals[max_val]))
print("max val @"+str(max_loc))
```

range() – creating int sequences

- generates pattern of numbers (arithmetic sequences only)
- to view the sequence immediately, call list()
- three arguments:
  - start (first value)
  - stop (sequence won't reach/pass this value)
  - step (how much to add each time)

```
>>> list ( range(0, 10, 1) )
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list ( range(0, 30, 5) )
[0, 5, 10, 15, 20, 25]
>>> list ( range( 0, 10, 3) )
[0, 3, 6, 9]
>>> list ( range(0, -5, -1) )
[0, -1, -2, -3, -4]
```

range()

more (common) ways to call:

- **range(start, stop)**
  - assumes step is +1
- **range(stop)**
  - assumes start is 0
  - assumes step is +1

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

Indexing in other orders

By constructing a different call to range(), we can index through our sequence in more sophisticated ways than just "in-order, all elements":

```
vals = [10,11,12,13,14,15,16,17]

for i in range(0, len(vals),2):
    print(vals[i])

for i in range(len(vals)-1, -1, -1) :
    print (vals[i])
```

watch out! using `range()`, you must get the indexes exactly right (never out of bounds). Slicing gracefully ignores out-of-bounds issues, indexing does not.

Nested Value Loops

- when we have multiple dimensions to our lists, we can use that many nested loops to access each item individually.
- Note the access pattern, as well as the total calculation.

```
xss = [[5,6,7],[8,9,10]]
total = 0
for xs in xss:
    for x in xs:
        print("\t+ "+str(x))
        total += x
print("total:",total)
```

output when run:

```
	+ 5
	+ 6
	+ 7
	+ 8
	+ 9
	+ 10
total: 45
```

Nested Index Loops

- Create an index for each dimension of your sequence.
- Nest loops for each dimension.
- Access each element individually (and starting from the entire structure like xss below), no matter how many dimensions.

```
xss = [[5,6,7],[8,9,10]]
for i in range(len(xss)):
    for j in range(len(xss[i])):
        print(xss[i][j])
```

output when run:

```
5
6
7
8
9
10
```

Nested Index Loops

- Our data doesn't have to have multiple dimensions for our algorithm to find use for nested loops.

```
# are there any duplicates in the list?
xs = [2,3,5,4,5,1,7,8]
has_dupes = False
for i in range(len(xs)):
    for j in range(len(xs)):
        if (i!=j) and xs[i]==xs[j]:
            has_dupes = True
            break
print("any dupes?",has_dupes)
```

```
# are there any duplicates in the list?
xs = [2,3,5,4,5,1,7,8]
has_dupes = False
for i in range(len(xs)):
    for j in range(i+1, len(xs)):
        if xs[i]==xs[j]:
            has_dupes = True
            break
print("any dupes?",has_dupes)
```

- note: what is different/better about the second version?

loop variable pattern-matching

- We can dissect each tuple with our for-loop variable(s).
- This is called tuple unpacking. Provide a pattern of variables.

```
tups = [('a',1), ('b',2),('c',3)]
for (c,n) in tups:
   print(c*n)
```

output when run:

```
a
bb
ccc
```

Aliases Example

```
xs = [1,2,3]
ys = [4,5,6]
both = [xs,ys]
xs[1] = 7
print("xs is",xs)
print("both is", both)
ys = [8,9]
print("ys is",ys)
print("both is", both)
```

program output:

```
xs is [1, 7, 3]
both is [[1, 7, 3], [4,5,6]]
ys = [8, 9]
both is [[1, 7, 3], [4, 5, 6]]
```

What is happening?

- variables are not the same as values.
- alias: when multiple names for the same location exist (such as **xs** vs **both[0]**) – changing the value by any name is witnessed from all others
- reassigning a variable re-establishes what the variable stores
- updating part of a value doesn't change which variables currently refer to the value
- We draw multiple arrows to the same value in our memory diagrams.

id( ) built-in function

- **id(thing)** returns a unique int value.
- detect aliases when **id(x)==id(y)** actual int value doesn't matter, only whether they are the same or not
- memory diagrams: two aliases both point to the shared value
- Note: python will auto-find/alias pre-existing strings!
- Note: Python also stores ints from -5 to 256.

```
>>> xs = [1,2,3]
>>> ys = [4,5,6]
>>> lists = [xs,ys]
>>> id(xs)
4302079040
>>> id(ys)
4301525288
>>> id(lists)
4301525360
>>> id(lists[0])
4302079040
>>> id(lists[1])
4301525288
>>> xs = [7,8,9]
>>> id(xs)
4301525864
>>> id(lists[0])
4302079040
```

Dictionaries – example

```
scores = {"Andrew":95, "Jerzy":82, "Mark":82 }
```

scores →

key	value
"Andrew"	95
"Jerzy"	82
"Mark"	82

dictionary:

- collection of key-value pairs
- no preserved ordering of keys ***(Python3.6+ is preserving insertion ordering!)***
- keys must be unique, keys must be hashable
- can add/update/remove key-value pairs
- great way to 'index' things by non-consecutive ints

Dictionary Examples

Syntax: `{key1:val1, key2:val2, … keyn:valn}`

```
empty        = { }
number_names = {1:"one", 2:"two", 3:"three"}
name_parts   = {"first":"George", "last":"Mason"}
random       = {1:"a", (1,2,3):"abc", None:"shall pass"}
```

Other Creation Strategies:

- Using dict function with keyword args, unquoted-strings as keys:

```
dict ( a=1, b=2, c=55 )
```

- Using dict function and sequence of length-two-sequences:

```
stuff = [["a",1], ["b",2], ["c",55]]
dict (stuff)
```

Dictionary Operations

function/method/operation	usage
len: # of key-value pairs.	len(d)
indexing: by key	d[k]
get: (use optional parameter ‘default’ if not found)	d.get(k) d.get(k, default)
del: remove a key-value pair	del d[k]
in, not in: test key's presence	k in d k not in d
clear: remove all key-value pairs	d.clear()
copy: create a shallow copy	d.copy()
keys, values, items: get the keys, values, or key-val pairs	d.keys() d.values() d.items()
pop: pop value at k (or return default) **popitem()**: pop any value	d.pop(k) d.pop(k,default) d.popitem()
update: insert all of another dict's key-value pairs	d_receiver.update(d_supplier)

Getting Keys, Values, or Key-Value Pairs

```
>>> d = {"a":1,"b":2,"c":3}
>>> d.keys()
dict_keys(['c', 'b', 'a'])
>>> d.values()
dict_values([3, 2,1])
>>> d.items()
dict_items([('c', 3), ('b', 2), ('a', 1)])
```

Dictionary Iteration

```
d = {"a":1, "b":2, "c":3}
print ("by keys:")
for k in d.keys():
    print (k, d[k])

print ("\nby values:")
for v in d.values():
    print (v)

print ("\nby items:")
for (k,v) in d.items():
    print (k,v)
```

→

```
by keys:
c 3
b 2
a 1

by values:
3
2
1

by items:
c 3
b 2
a 1
```

Dictionary Ideas

- Make a sparse matrix with int-tuple keys.
  - Only stores needed keys
  - Guaranteed no duplicate entries for same key
  - Main issue: navigating in order without attempting

```
mymap = {(0,0):"origin",
         (1,3):"secret lair",
         (1000,32):"forgotten isle"
        }
```

Exceptions

Exceptions

Some Common Exceptions

Exception Type	Description
FileNotFoundError	tried to open a non-existent file
IndexError	tried to index into a structure with a not-present index.
KeyError	tried to access non-existent key in a dictionary.
NameError	identifier for a name couldn't be found in scope.
SyntaxError	syntax error encountered.
TypeError	type error encountered, e.g. argument to built-in is of wrong type.
ValueError	built-in function/operation received value of right type, but wrong value (e.g., int() received a str, but it didn't represent a number)
ZeroDivisionError	tried to divide by zero.

Exceptions Hierarchy (excerpt)

- There are many exception classes organized into a hierarchy
  → each name here is its own python type!
- indentations: indicates inheritance (parent/child relationships).
  - A KeyError is a more specific kind of LookupError; it's allowed anywhere a LookupError is.
- abbreviated version of the hierarchy → → → → → →

```
BaseException
+-- KeyboardInterrupt
+-- Exception
     +-- ArithmeticError
     |    +-- ZeroDivisionError
     +-- EnvironmentError
     |    +-- OSError
     |         +-- FileNotFountError
     +-- EOFError
     +-- LookupError
     |    +-- IndexError
     |    +-- KeyError
     +-- NameError
     +-- SyntaxError
     +-- SystemError
     +-- TypeError
     +-- ValueError
```

(found at https://docs.python.org/3/library/exceptions.html#exception-hierarchy)

variations: multiple except blocks, multiple types per block

ex3.py

```
try:
    import sys
    filename = sys.argv[1]
    f = open(filename)          # file might not exist
    lines = f.readlines()
    f.close()
    xs = []
    for line in lines:
        xs . append ( int(line) )# might not be an int.
    secret = xs[3] /  xs[10]        # index might not exist
    print("secret result:",secret)
except FileNotFoundError:
    print("file didn't exist.")
except (ValueError, IndexError) as e:
    print("bad input!", type(e), str(e))
except Exception as anyname:    # any zero-division??
    print("catch-all: unforseen! ", str(anyname))
print("end of example.")
```

Notes

- when exception occurs, **only the first compatible except-block runs**!

Example details

- **sys.argv** lets us access the command-line arguments
- except block for **FileNotFoundError** didn't want to inspect the exception value; no **as e** clause needed.
- **ValueError**, **IndexError**, and any child classes of exceptions are all handled here.
    - wanted to inspect the object, so **as *<name>*** clause included.
- ***except Exception*** block handles exceptions of type **Exception** and any child classes – that's all exceptions!

variations

- we can inspect the exception value if desired:

```
except SomeType as anyname:
    statements <can use anyname>
```

- we can ignore the particular value and still catch those types by skipping the as-clause:

```
except SomeType:
    statements
```

variations

- we can have multiple **except** blocks.
  - first block to handle the actual type of raised exception is the only one to run
  - "parent" types of exceptions match all child types (the deeper indentations of that chart are child types)
  - **except Exception** thus catches anything

- we can catch anything, and ignore the particular value, with a raw **except:** block

caution – things can still crash

- any raised exception whose type isn't compatible with any of the except blocks will "escape":
  - it crashes further, out of the next layer of try-blocks, function calls, until it either is caught elsewhere or crashes the entire program.

ex4.py

```
xs = [5,10,15,20]
try:
    index = int(input("which spot? "))
    val = xs[index]
    print("you chose "+str(val))
except ValueError:
    print("that wasn't an int.")
print("end of program.")
```

Sample Inputs:

- 2 # successful
- three # ValueError caught
- 39 # IndexError escapes!

Validating Input

Loop continues to execute, raising and handling exceptions, until user complies.

ex5.py

```
need_input = True
while need_input:
    try:
        n = int(input("#items: "))
        fr = int(input("#friends: "))
        each = n/fr
        need_input = False
    except Exception as e:
        print(e)
print("everyone gets %s items." % each)
```

sample calls

```
demo$ python3 ex5.py
#items: asdf
invalid literal for int() with base 10: 'asdf'
#items: 3
#friends: 0
division by zero
#items: 10
#friends: 5
everyone gets 2.0 items.
```

validating input: alternate version

We can use **while True:** and **break** with exceptions for a convenient way to escape: if any exceptions occur, we skip the break and the loop forces us to try again.

ex5.py

```
need_input = True
while need_input:
    try:
        n = int(input("#items: "))
        fr = int(input("#friends: "))
        each = n/fr
        need_input = False
    except Exception as e:
        print(e)
print("everyone gets %s items." % each)
```

ex5_alt.py

```
while True:
    try:
        n = int(input("#items: "))
        fr = int(input("#friends: "))
        each = n/fr
        break
    except Exception as e:
        print(e)
print("everyone gets %s items." % each)
```

Practice Problem

What happens if we instead had the while loop inside the try block, like this?

practice1.py

```
try:
    need_input = True
    while need_input:
        x = int(input("#: "))
        need_input = False
except Exception as e:
    print(e)
print ("successfully got x: "+str(x))
```

Raising Exceptions

- We can generate an exception on purpose (and hopefully catch it somewhere else!)
- performed with a **raise** statement, which needs an expression of some Exception type. This usually means calling a constructor (__init__ method). Examples:

  - `raise Exception("boo!")`
  - `raise ArithmeticError ("this doesn't add up!")`
  - `raise ValueError("needed a positive number")`

  - ```
    except IOError as e:
        print ("catching it, re-raising it.")
        raise e
    ```
    ```

Functions

Function definitions

- Function definition statement: given a name, parameters list, and body.
- Can be nested to any depth!
- Functions are first-class: they can be passed around as values (uncalled!)

```
def max3(a, b, c):
    if a>=b and a>= c:
        return a
    if b >= c:
        return b
    return c
```

Argument options

- **Positional arguments**
  - Plain parameters; must be given, no defaults available.
  - def'n: **def foo(a, b, c, d)**
  - call: foo(1,2,3,4)
- **Default arguments**
  - At definition site; tail of params list can have defaults.
  - def'n: **def foo (a, b, c=0, d=0, go=True)**
  - calls: foo(1,2) foo(1,2,3) foo(1,2,3,False)
  - Beware of complex default values!
    - def put_stuff (these_vals, here = [])
- **Keyword arguments**
  - At call time. Supply arguments in any order, by name.
  - powerful when mixed with default args.
  - **foo (b=5, a=100, go=False)** # note: c not given, will use default

Argument Options – variadic arguments

- **Variadic arguments**: accepting any # of positional args.
  - at def'n: last positional arg has * in front; grouped into tuple.
    - def not_first(drop_me, *the_rest): return the_rest
  - at call: feed any number.
    - not_first("drop",1,2,3)   not_first("give none")
  - Related: use sequence to feed multiple regular (non-variadic) arguments:
    - "explode" the sequence with a star: *xs
    - xs = [2,4,6]; max3(*xs)

Argument Options – variadic keyword arguments

- **Keyword args:**
  - at def'n: last arg has ** in front
    - can give *arbitrary keyword arguments*
    - grouped into dictionary.
    - .format() usage: "{:>{padwidth}}".format(value, padwidth=max(len(a),len(b)))
  - Related: explode a dictionary into plain keyword args
    - d={'a':2,'b':4,'c':6}; max3(**d)

- **Mixing Styles:**
  - All positional args, then variadic args, then default args, then kwargs.
  - def foo(pos, itio, nal, *vargs, d="ef",au="LTS", **kwargs)

File I/O

Reading files - examples

```
file_ref = open ("sample.txt")          # get access to file
str_contents = file_ref . read ()       # read entire file as string
file_ref . close ()                     # close file when done

<use entire file's contents as string>
```

```
fileRef = open ("myfile.txt")           #access it
lines = fileRef . readlines( )          #read it to list of line-strings
fileRef . close( )                      #always close the file!
for line in lines:                      #use it! just a list of strings now
    print(line.upper(), end="")         # shout it out!
print ("done!")
```

Reading files - more examples

```
f = open ("sample.txt")              # get access to file
for line in f:                       # lazily read a line at a time
    print(line.upper(), end="")      # shout it out!
f.close()                            # always close the file!
```

```
with open ("myfile.txt") as f:       # will auto-close f after with-scope.
    for line in f:                   #use f. again a line at a time here.
        print(line.upper(), end="")  # shout it out!
```

reading files

- compare to physically reading a book
  - also, think of file's contents as a python string, indexed 0 and up
- you have a 'bookmark' tracking where you are, updating as you go
  - you can read a few characters at a time, a line at a time, or the whole thing at once

sample call	meaning (starting at your 'bookmark' always)	return value
f . read (n)	read up to n characters	string
f . read ( )	read all remaining characters	string
f . readline (n)	read up to n characters on current line	string
f . readline ( )	read the rest of the current line	string
f . readlines ( )	read all remaining lines	list of strings

write(x)

test.txt (before close)

```
0123456789
ABCDEFGHIJ
qrstuvwxyz
```

```
>>> file = open("test.txt","w")
>>> s = "Line 1\nLine 2\nLine 3\n"
>>> file.write(s)
21
>>> file.close()
```

test.txt (after close)

```
Line 1
Line 2
Line 3
```

- absolutely nothing except what you write goes into the file – no newlines, spacings, or anything.
- you can write parts of one line in as many write calls as you need

file writing methods

- calling write and writelines is like successive print calls, only the output goes to a file (and no newlines or separators are ever added, only exactly what you write)
- nothing is actually written to the file until you close it!

method	behavior	example call
write(x)	writes string x to file	f.write("stuff\nhere")
writelines(xs)	writes strings in list xs to file	parts=['a\n','\b'n\c', 'd','e','f'] f.writelines(parts)

Classes

Python classes – Initial Thoughts

- All good hygiene is your responsibility.
  - Subclasses can pretty much replace/ignore parent class
  - add/delete instance variables at will, per object
  - add/delete methods at will, per class

- Double-underscore naming convention: "special" methods that hook into builtin functions or class implementation will have double-underscores at beginning/end of names:

```
__init__    __str__    __repr__
__eq__      __lt__     __gt__       ...
```

Class Definitions

```
class Person:
  min_age = 0  # class variable: shared; Person.min_age
  # constructor.
  def __init__(self, name, age):
    self.name = name
    self.age = max(age, min_age)
  def greet(self):      # note the \ arbitrary line break
    return ("Hi, I'm {}. I'm {} years old." \
            .format(self.name, self.age))
```

SubClass Definitions

```
class Point:
  def __init__(self, x=0,y=0):
    self.x = x
    self.y = y
  def magnitude(self):
    return (x*x + y*y) ** 0.5
  def __str__(self):
    return f"({self.x},{self.y})"
  def shift(xshift=0, yshift=0):
    self.x += xshift
    self.y += yshift
  def slope(self, other):
    rise = other.y - self.y
    run = other.x - self.x
    return rise / run
```

```
class Point3D(Point):
    def __init__(self, x, y, z):
        # manually call parent's __init__
        # preferably first!!
        super().__init__(x,y)
        self.z = z
    def __str__(self):
        return "(%s,%s,%s)" % (self.x, self.y, self.z)
    def __repr__(self):
        return str(self)
    def project_down(self):
        # go to 2D; make new object.
        return Point(self.x, self.y)
```

Class oddness: adding fields to individual objects

- You can (accidentally?) create new instance variables for any single object by assigning it.

```
p = Point(1,2)
p.another = 3

del p.another
del p.x
```

- You can (accidentally?) add/modify methods for a class:

```
# wishing you had this in Point3D? Make it happen!
def shift3(self, xshift=0, yshift=0, zshift=0):
   self.x+=xshift; self.y+=yshift; self.z+=zshift

Point3D.shift3 = shift3 # now available!
tri.shift3(100,200,300) # hurts my brain…
```

User-Defined Exceptions

- We can create our own types of exceptions.
- They can be raised, propagated, and caught as usual.
- Just include Exception as the parent class as shown below

```
class BadInput(Exception):
    def __init__(self, value):
        self.value = value
```

```
x = int(input("#? "))
  if x==13:
    raise BadInput("that's unlucky!")
  print(x*10)
except BadInput as e:
  print("uhoh: "+e.value)
except Exception as e:
  print(e)
```

sample usage

```
demo$ python3 ex11.py
#? 5
50
demo$ python3 ex11.py
#? 13
uhoh: that's unlucky!
demo$ python3 ex11.py
#? asdf
invalid literal for int() with base 10: 'asdf'
```

Functions and Recursion

Recursive functions

- Just call yourself:

```
def fact(n):
  if n<=1:
    return 1
  return n*fact(n-1)
```

- Or call each other:

```
def even(n):
  if n==0:
    return True
  return odd(n-1)

def odd(n):
  if n==0:
    return False
  return even(n-1)
```

Higher-order functions

- map, zip, and reduce: common functions that need functions as arguments.
- **map**: given a function and a list of values, apply the function to each item and generate a list of answers.
  - map(lambda s : s.upper(), ["Hello", "reu"]) ⟶ ["HELLO", "REU"]
- **zip**: given two lists, create list of pairs, combining same-index values from each. Only as long as shortest list.
  - zip([1,2,3],[4,5,6,7,8]) ⟶ [(1,4),(2,5),(3,6)]
- **functools.reduce**: given binary operator and list, collapse list to single value.
  - reduce(lambda x, y: x+y, [1,2,3,4,5]) ⟶ 15

Using higher-order functions

- Average test scores from all students whose attendance is poor:

```
mia = filter(lambda s: s.attendance<0.25, roster)
mia_scores = map (lambda s: s.test, mia)
avg_mia_score = sum(mia_scores)/len(mia_scores)
```

- Implement other things:

```
def max(xs):  return reduce(lambda x,y: x if x>y else y, xs)
def plus_ones(xs): return list(map(lambda x: x+1, xs))
def evens(xs): return filter(even, xs)
```

Speed Up and Go Big!

Dealing with more data, deeper recursion, etc.

Recursive functions – memoize to go faster

- Save answers in shared mutable default value:

```
def fib(n):
 if n<=1:
   return 1
return fib(n-1)+fib(n-2)
```

```
def fastfib(n, ans={}):
 if n<=1:
   return 1
 n1 = ans[n-1]  if  (n-1) in ans  else fastfib(n-1)
 n2 = ans[n-2]  if  (n-2) in ans  else fastfib(n-2)
 ans[n] = n1+n2
 return ans[n]
```

Out of stack space? Make your own.

```
fib(123456) # RecursionError, booo! Now what?
```

- Build your own stack. (see **def iterfib** in tutorial.py)
- rethink your whole algorithm. (see **def loopfib** in tutorial.py)