Recall: Factorials

3! = 3 \times 2 \times 1
4! = 4 \times 3 \times 2 \times 1

More generally,
\[ n! = n \times (n-1)! \quad \text{# recursive rule} \]
Factorials have a recursive definition

\[ 3! = 3 \times 2 \times 1 \]
\[ 4! = 4 \times 3 \times 2 \times 1 = 4 \times 3! \]

More generally,

\[ n! = n \times (n - 1)! \quad \text{# recursive rule} \]
\[ 0! = 1 \quad \text{# base case} \]

```python
def factR(n):
    '''problem has recurring pattern → recursive solution'''
    if n == 0:
        # base case
        return 1
    else:
        # recursive call
        return n * factR(n-1)

>>> factR(3)
```

```python
def factI(n):
    '''not as elegant
    also less expensive'''
    result = 1
    for i in range(1, n+1):
        result *= i
    return result
```
Why recursion?

- elegant approach to problem solving for problems with a recursive structure
- elegant approach to writing programs – underlying algorithm is clear – solutions are simpler to write, analyze, and understand

Why not?

- can be prohibitively expensive

---

Finishing / Starting

✓ Recursion, cont’d
- List comprehensions
- Intro to user-defined classes
- Programming environments
- Summing up

Final Exam Review – all labs (Q/A)

“When you express your understanding in code, you debug your brain.”

---

List Comprehensions (declarative style programming)

Given:

\[ S = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\} \]

Generate:

\[ T \text{ is a list of } x \text{ such that } x \text{ is a member of } S \text{ and } x \text{ is even} \]
CIS 210
List Comprehensions

S = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

T is a list of x such that x is a member of S and x is even

\{x \mid x \in S \text{ and } x \text{ is even}\} \quad \#\text{mathematical}

T = [] \quad \#\text{procedural}
for x in S:
    if even(x):
        T.append(x)

T = [x for x in S if even(x)] \quad \#\text{declarative (filter pattern)}

---

def mode(alist):
    '''(list) -> list
    countd = genFreqTable(alist)
    countli = countd.values()
    maxct = max(countli)
    modeli = []
    # there may be more than one mode
    for k in countd:
        if countd[k] == maxct:
            #another filter pattern
            modeli.append(k)
    return modeli

modeli = [k for k in countd if countd[k] == maxct]
List comprehensions are a concise way to create lists. The general syntax is:

```
[<expression>
for <item> in <sequence> ...  # other collection(s)
if <condition>]
# filtering (if needed)
```

```
modeli = [k for k in countd if countd[k] == maxct]
```
List comprehensions are a concise way to create lists. The general syntax is:

```
[<expression>
 for <item> in <sequence> ...
 if <condition>]
```

Each item in the new list is the result of applying a given operation `<expression>` to a value `<item>` from a sequence `<sequence>`.

```python
>>> [2 * i for i in [1, 2, 3]]
??
```

Replace the `??` with the result of executing the following code (indicate 'error' if the result would be a Python error message):

```python
>>> factors = [x for x in range(1,10) if x % 3 == 0]
>>> factors
??
```

Replace the `??` with the result of executing the following code (indicate 'error' if the result would be a Python error message):

```python
>>> name = "University of Oregon"
>>> name = name.split()
>>> new = name[0][0] + name[2][0]
>>> new
??
```

---

```
def addlist(m, n):
    """(list, list) -> None
    Exam function.
    ""
    for i in range(len(m)):
        if i < len(n):
            m[i] += n[i]
    return None

def q9(x, y):
    """(list, list) -> None"
    z = x
    addlist(x, y)
    # more advanced IDE
    print(x); print(y)
    w = []
    # rewrite as list comprehension
    for item in y:
        w.append(item / 10)
    print(w)
    return None
```

---

**CIS 210**

**Finishing / Starting**

- Recursion, cont’d
- List comprehensions
  - Intro to user-defined classes
  - Programming environments
  - Summing up

**Final Exam Review** — all labs (Q/A)

_Everybody should learn how to program a computer, because it teaches you to think._
Recall: Python data types (objects):

- **type** – range of values and operations
- **value(s)**
- **id**

CIS 210

A **type**, or **class**, is a template, for objects.

**Objects** are **instances** of a class.

Every Python object is an instance of a class:

```python
>>> y = str()
>>> type(y)
<class 'str'>
>>> y
''
>>> id(y)
4298926768
```

- **str** is a **constructor method** for class **str**
- **str() instantiates** a **str object**
- **y** is an **instance (object)** of class (type) **str** with **value** and **id**
class Turtle

>>> t1 = Turtle()
>>> t2 = Turtle()

• Turtle is a constructor method for class Turtle

• Turtle() instantiates a Turtle object

• t1 and t2 are instances of the class Turtle (turtle objects)

Recall: Python data types (objects):

- type
  - range of values →
  - attributes
  - object methods (operations)
  - object descriptors

- value(s)
- id

>>> t1.heading()
>>> t1.pos()
>>> t2.heading()
>>> t2.pos()
>>> t1.color()
>>> t2.shape()

>>> t1.fd(100)
>>> t2.fd(200)
>>> dir(Turtle)
>>> dir(t1)

>>> t1.heading()
>>> t1.seth(75)

>>> t1.pos()
>>> t1.setpos(0, 0)

>>> t2.heading()
>>> t1.shape('turtle')

>>> t2.pos()
>>> t2.color('blue')

>>> t1.color()

>>> t2.shape()
CIS 210

Classes and Objects

special constructor method __init__ initializes class names and values:
  for example, x and y

class Point():
    """Represents a point in 2-d space."""
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y

>>> p1 = Point()
>>> p2 = Point(3, 4)

>>> p3 = Point(99, 100)
>>> p3
<__main__.Point object at 0x105518da0>

>>> type(p3)
<class '__main__.Point'>

>>> id(p3)
4384198048

>>> dir(p3)       >>> p3.x       >>> p3.y
[... 'x', 'y']   99           100

Add methods to get and set (mutate) attribute values:

class Point(object):
    """Represents a point in 2-d space."""
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y
    def getx(self):
        return self.x
    >>> p2.getx() 4
    def gety(self):
        return self.y
    >>> p2.gety() 4
    >>> Point.gety(p2) 4
Classes and Objects

```python
>>> p1 = Point()
>>> p2 = Point(3, 4)
>>> p1.getx()   >>> Point.getx(p2)
0             3
>>> p1.sety(100) >>> p1.gety()
100
```

class Point(object):
    '''Represents a point in 2-d space.'''
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y
    def getx(self):
        return self.x
    def setx(self, newx):
        self.x = newx
        return None
    def gety(self):
        return self.y
    def sety(self, newy):
        self.y = newy
        return None

```python
>>> p1 = Point()
>>> type(p1)
<class '__main__.Point'>
>>> p2 = Point(3, 4)
>>> id(p1)
4389818152
>>> p1.getx()   >>> Point.getx(p2)
0             3
>>> p1.sety(100) >>> p1.gety()
100
```

Other special methods: `__str__` and `__add__`, for example

```python
>>> ctr = 1
>>> ctr.__add__(1)    >>> ctr + 1
>>> s = 'abc'
>>> s.__add__('def')  >>> s + 'def'
```

`__add__` is a special method used by the `+` operator, its behavior varies per object type.
Classes and Objects

Other special methods:

`__str__` and `__add__`, for example

```python
>>> ctr = 1
>>> ctr = ctr.__add__(1)  # This is equivalent to ctr + 1

`__str__` is a special method where the object returns a string representation of the object

this is what the print function uses
```

```python
class Point(object):
    '''Represents a point in 2-d space.''

    def __str__(self):
        return f'***{self.x}, {self.y}***'
```

```python
>>> p4 = Point(0, 100)
>>> print(p4)
*** 0, 100 ***
```

Finishing / Starting

✓ Recursion, cont’d
✓ List comprehensions
✓ Intro to user-defined classes

• Programming environments
• Summing up

Final Exam Review – all labs (Q/A)

Everybody should learn how to program a computer, because it teaches you to think.
- **Editor** for writing programs in a particular language and according to style guidelines

- **Translator** (interpreter/compiler, linker) for executing program (and reporting results)

- **Other tools**, e.g., for testing and debugging

---

- An integrated development environment (IDE) is a software application that provides comprehensive facilities to computer programmers for software development, e.g.,
  - source code editor that supports writing programs
  - Shell (compiler, interpreter)
  - debugger
  - version control system
  - class browser, an object browser, and a class hierarchy diagram, for use in object-oriented program development
  - tools to manage GUI interface development

---

- Individual tools also exist outside of IDEs:
  - Text editor + Command line + Python debugging module (for example)
Once upon a time...

switches

cards

teletype machines (command-line)

video display terminals (command-line)

Graphical User Interfaces (GUIs – point and click)

**command line** interface programs (Terminal/cmd.exe)

---

**Integrated Development Environment (IDE) v. Command-line programming**

**command line** - shell* for operating system

text-based (pre-GUI)

*program that accepts commands as text input and translates them to appropriate functions in an underlying program, e.g., interpreter, OS

---

**Integrated Development Environment (IDE) v. Command-line**

sysadmins-air:~ kfreeman$ **python3.6**
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 03:03:55) [GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print('hello, world')
hello, world
>>> quit()
sysadmins-air:~ kfreeman$
Recall:
...

def main():
    """Square root comparison program driver."""
    sqrt_compare(25, 5)
    sqrt_compare(25, 10)
    ...
    return None

sysadmins-air:F18-projects kfreeman$ python3.6 p22_sqrt_key_W19.py
For 25 using 5 iterations:
  mysqrt value is: 5.000023178253949
  math lib sqrt value is: 5.0
  This is a 0.0 percent error.
For 25 using 10 iterations:
  mysqrt value is: 5.0
  math lib sqrt value is: 5.0
  This is a 0.0 percent error.
...

sysadmins-air:W19-projects kfreeman$ python3.6 p22_sqrt_key_W19.py
For 25 using 5 iterations:
  mysqrt value is: 5.000023178253949
  math lib sqrt value is: 5.0
  This is a 0.0 percent error.
...

sysadmins-air:W19-projects kfreeman$ python3.6 -i p22_sqrt_key.py
For 25 using 5 iterations:
  mysqrt value is: 5.000023178253949
  math lib sqrt value is: 5.0
  This is a 0.0 percent error.
...

sysadmins-air:W19-projects kfreeman$ python3.6 -i p22_sqrt_key.py
For 25 using 10 iterations:
  mysqrt value is: 5.0
  math lib sqrt value is: 5.0
  This is a 0.0 percent error.
...

sysadmins-air:W19-projects kfreeman$ python3.6 -i p22_sqrt_key.py
For 25 using 10 iterations:
  mysqrt value is: 5.0
  math lib sqrt value is: 5.0
  This is a 0.0 percent error.
**Integrated Development Environment (IDE) v. Command-line**

I. (a) **def main():**
   
   ```python
   '''Square root comparison'''
   with open('test.txt') as t:
       tests = int(t.readline())
       for i in range(tests):
           line = t.readline()
           n = int(line[0])
           r = int(line[1])
           sqrt_compare(n, r)
   return None
   ```

II. (b) **def main():**

   ```python
   def main(n, r):
       sqrt_compare(n, r)
   ```

   ```python
   import sys
   def main():
       '''Use arguments supplied at command line'''
       print(sys.argv)  # what is sys.argv?
       n = int(sys.argv[1])
       r = int(sys.argv[2])
       sqrt_compare(n, r)
   return None
   ```

III. (c) **def main(n, r):**

   ```python
   import sys
   def main():
       '''Use arguments from a file: p22_wk10demo'''
       $python3.6 p22_sqrt_key.py < p22_wk10demo.txt
   ```

   ```python
   num_tests = int(input())  # first line of file is number of tests
   for i in range(num_tests):  # form the list
       testli = input().split(',')
       n = int(testli[0])
       r = int(testli[1])
       sqrt_compare(n, r)
   return None
   ```

IV. (d) **def main():**

   ```python
   def main():
       with open('test.txt') as t:
           tests = int(t.readline())
           for i in range(tests):
               line = t.readline()
               n = int(line[0])
               r = int(line[1])
               sqrt_compare(n, r)
   return None
   ```

---

**kfreeman$ python3.6 p22_sqrt_key.py 81 10**

```python
import sys
def main():
    num_tests = int(input())
    for i in range(num_tests):
        testli = input().split(',')
        n = int(testli[0])
        r = int(testli[1])
        sqrt_compare(n, r)
    return None
```
Integrated Development Environment (IDE) v. Command-line

**Writing – executing – testing and debugging code:**

**IDE:** GUI, integrated, handy tools in one place, overhead to learn (and then learn again). Great for development and testing, especially for large projects.

**command line:** Shell (text only), not integrated, portable, powerful

built-in editors: e.g., emacs, vim, TextEdit, IDLE

---

**CIS 210 / Welcome**

**What can you expect from CIS 210**

**CIS 210 Focus:** Computational Problem Solving

• Understand/revise/refactor/create/implement algorithms and code

• Expand/improve software development skills

• Demonstrate robust mental models of data representation and code execution

• Gain familiarity with a sampling of classic computer science problem domains/asgorithms.

---

**CIS 210 / Welcome**

**Computational problem solving**

is an approach to problem solving that is inspired and constrained by the possibilities and limitations of computers and computing

---

**CIS 210**

**Finishing / Starting**

✓ Recursion, cont’d

✓ List comprehensions

✓ Intro to user-defined classes

✓ Programming environments

• Summing up

Final Exam Review – all labs (Q/A)

*Everybody should learn how to program a computer, because it teaches you to think.*
CIS 210 / Welcome

Computational Problem Solving

TASK/PROBLEM ➔ Computational Thinking ➔ PRECISE SPECIFICATION OF PROBLEM/SOLUTION/ALGORITHM

SOLUTION/ALGORITHM ➔ Design/Coding ➔ COMPUTER PROGRAM

COMPUTER PROGRAM ➔ Testing/Debugging ➔ HIGH QUALITY COMPUTER PROGRAM

HIGH QUALITY COMPUTER PROGRAM ➔ Execute ➔ AUTOMATIC, FAST, RELIABLE, REUSABLE SOLUTION

CIS 210

understand, develop, implement, and algorithms for computational problem solving; use structured design and testing methods to develop and implement programs

problem: review, clarify, write examples

algorithm: review, revise, or create algorithm to solve the problem – pseudocode, check against examples

program: design (top down) – determine functions for each high level function:

problem, algorithm – review, clarify, ...

write a brief description
write function header
write type contract
write test cases (include examples)
write the return statement

CIS 210

understand, develop, implement, and algorithms for computational problem solving; use structured design and testing methods to develop and implement programs, cont’d.

programming: coding (bottom up)

review Python toolkit
write Python code (use pseudocode to start)

CIS 210

understand, develop, implement, and algorithms for computational problem solving; use structured design and testing methods to develop and implement programs, cont’d.

programming: testing and debugging

test code using examples of use (automated testing)
revise and retest as needed
testing

• cases for basic, edge, for various expected arguments and results
errors

• syntax, runtime, logical, documentation
• assert and try/except
CIS 210 / Welcome

demonstrate robust mental models of data representation and code execution

Python is a language (virtual machine) AND
Python is a program (translator/interpreter)

CIS 210

demonstrate good understanding of a high level programming language (Python), for example:

- assignment statement
- expressions
- Python repetition – for, while
- Python conditionals – if
- numeric data types (int, float) and operations (e.g., +, **, round, abs)
- Boolean data type and operations (e.g., &&, ||, and)
- string data type and operations (e.g., x, len, count, find, format, formatted strings)
- Python collections data types and operations – tuples, lists, dictionaries; comprehensions
- Python files and file processing
- data type "coercion" functions (e.g., str, int, list, float)
- NoneType (None)
- print/input
- Python Standard library – math, turtle, random modules; import (if __name__ == '__main__')
- user-defined functions; function design; docstrings
- IDE interactive development environment; intro to programming environments
- Python introspection: help, dir, type, id, isinstance, au gc, au chdir
- run-time checking of data and code; assert; try/except

CIS 210

familiarity with sampling of classic computer science problem domains and algorithms

- Monte Carlo algorithms
- encoding and decoding
- recursion; iterative and recursive algorithms
- binary representation of symbols
- data analysis (access, process, report)
- data mining (k-means cluster analysis) / visualization
- binary search (big O notation)
- loop patterns (accumulator/reduce, map, filter)
CIS 210

Finishing / Starting

✓ Recursion, cont’d
✓ List comprehensions
✓ Intro to user-defined classes
✓ Programming environments
✓ Summing up

Final Exam Review – all labs (Q/A)

Everybody should learn how to program a computer, because it teaches you to think.

CIS 210

Python/programming toolkit so far

assignment statement
expressions
Python repetition – for, while
Python conditionals – if
numeric data types (int, float) and operations (e.g., +, **, round, abs)
Boolean data type and operations (e.g., <, and)
string data type and operations (e.g., +, len, count, find, format, formatted strings)
Python collections data types and operations – tuples, lists, dictionaries, comprehensions
Python files and file processing
data type “exercise” functions (e.g., str, int, list, float)
NoneType (None)
print/input
Python Standard Library – math, turtle, random modules; import (if __name__ == '__main__')
user-defined functions; function design; docstrings
IDE interactive development environment; intro to programming environments
Python introspection: help, dir, type, id, isinstance, os.getcwd, os.chdir
run-time checking of data and code: assert; try/except

CIS 210

Programming/Software Engineering/Computer Science Concepts

Computational Problem Solving: designing, implementing, checking, revising algorithms/programs.
Good programming style: function docstrings (type contract; description including parameters, returned value, and side effects if any; examples of function use); well-named variables, use of whitespace between operators and sections of code, judicious use of inline comments (why not what).
Python is a programming language and Python is an interpreter (program)
Python Shell is a REPL (read-evaluate-print loop)
Python primitive elements: Objects - value/attributes, type, memory location (id)
Combining primitive elements: Expressions - expressions evaluate to a value; overloaded operators; methods/functions/operators; short circuit evaluation of boolean expressions
What happens when an assignment statement is executed: assignment statements are not expressions and do not return a value; memory allocation; reference semantics; namespaces – builtins, global (__main__ and modules), and local; variable scope; dynamic typing
Other language considerations – strong typing, mutable and immutable data types
Functions are an executable data type; what happens when a function is called:
Activation record/stack frame added to call stack for local namespace; return address
Call-by-assignment parameter passing
Functions always return a value (sometimes None)
Functions sometimes have side effects
Functions can be passed as arguments

CIS 210


Executing and revising functions - hand-tracing program execution; hand-tracing function execution; program function diagrams; refactoring functions
Binary representation of integers and characters
Systematic approaches to testing and debugging; automated testing
Recursion – problem-solving approach; implementing and tracing recursive code
Algorithms – iterative and recursive algorithms; accumulator, map, filter patterns; Monte Carlo algorithms; binary representation and conversion; encryption and decryption; data analysis and data mining (k-means cluster analysis); binary search (and big-O notation)
CIS 210 Learning Outcomes

- understand, develop, implement algorithms for computational problem solving;
- use structured design and testing methods to develop and implement programs;
- read, write, revise, document, test, and debug code;
- demonstrate robust mental models of data representation and code execution;
- demonstrate good understanding of a high level programming language;
- introduce and/or implement a sampling of classic computer science problem domains and algorithms.