Welcome to CIS 210: Computer Science I. This course covers basic concepts and practices of computer science. Topics include algorithmic problem solving, levels of abstraction, object-oriented design and programming, software organization, analysis of algorithm and data structures.

Prerequisites: MATH 112. Prior programming experience strongly encouraged.

NOTE: IF YOU ARE HAVING TROUBLE VIEWING THE COURSE WEBSITE, TRY CONNECTING TO UO SECURE RATHER THAN UO GUEST. I AM LOOKING INTO THIS.

Course Website: https://classes.cs.uoregon.edu/20W/cis210/
Class Encore: Small study groups for challenging classes allow students to gather once a week outside of class to practice course concepts and strategies. Peer leaders attend the classes and design collaborative activities for each group meeting. Participation is FREE and open to ALL students enrolled in the class.

Encore Website: https://classencore.uoregon.edu
Python toolkit so far

numeric data types (int, float) and operations (e.g., +, %, **, pow, round, abs)
[Boolean data types]
[strings]
expressions

variables (identifiers)
assignment statement
Python repetition – while

user-defined functions: def, parameter list, docstring, code, return
IDLE interactive development environment
Python introspection – help, dir functions

[tools in braces are in project 11-quiz example code only]
Computer Science concepts

Computer Science overview: support for Computational problem solving
Computational Problem Solving: designing, implementing, checking, revising algorithms/pgms.

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); abstraction
Python Shell is a REPL (read-evaluate-print loop)
Programming language basics: keywords, primitive elements, identifiers
Python primitive elements: Objects - value/attributes, type
Combining primitive elements: Expressions - expressions evaluate to a value
Naming values: Variables/assignment - assignment statements are not expressions and do not return a value; namespaces – builtins and __main__
Functions are an executable (callable) data type
CIS 210 Computer Science I

Does this code do what it is supposed to do?
→ CIS 210 style guidelines (programming best practices)

Focus on functions
-- why functions?
-- what happens when a function is called?
-- activation record/call stack
-- Python parameter passing
-- functions are small programs
-- functions always return values
-- functions may cause side effects - Python print
-- functions can call functions (lab, too)

Python Standard Library (importing modules) (lab, too)

Accumulator pattern and Python repeat - for

A structured approach to Project 2-2  Turtle graphics (lab only)
Does this code do what it is supposed to do?

def et(i,n):
    sd = 10000
    t = (i-sd-(4150*n))*.2
return t
Developers, Despair: Half Your Time Is Wasted on Bad Code

A study by online payments firm Stripe found 31% of developer time is wasted on routine maintenance work, draining $300 billion each year from global GDP.

The survey of 1,000 developers and 1,000 C-level executives found that on average, about half of a developer's work time is devoted to maintenance activities such as debugging, modifying, and fixing code. Developers reported work 41 hours per week on average, including 17.3 hours on maintenance work.

Each of the estimated 18 million developers in the world contributes $51,000 to global GDP annually, totaling $918 billion worldwide, according to Stripe. More efficient use of developer time could raise global GDP by $3 trillion over the next decade, Stripe says.

The leading obstacles to productivity cited by developers were maintenance of legacy systems, leadership's prioritization of projects, and building custom technology.
Developers, Despair: Half Your Time Is Wasted on Bad Code

By ZDNet
September 14, 2018
Comments

A study by online payments firm Stripe found 31% of developer time is wasted on routine maintenance work, draining $300 billion each year from global GDP.

The survey of 1,000 developers and 1,000 C-level executives found that on average, about half of a developer's work time is devoted to maintaining code, and about one-quarter to writing new code. The remaining one-quarter is spent working on other projects.

Each year, developers waste more than a billion hours on code maintenance. The costs range from millions to billions, depending on the size of the project.

From ZDNet
View Full Article

Abstracts Copyright © 2018 Information Inc., Bethesda, Maryland, USA
Does this code do what it is supposed to do?

Generate an estimate for federal income tax based on reported income and number of exemptions. The standard deduction ($10,000) and standard exemption ($4,150) are always used and the tax rate is assumed to be 20%. Return the estimated tax.

```
def et(i,n):
    sd = 10000
    t = (i-sd-(4150*n))*.2
    return t
```
Generate an estimate for federal income tax based on reported income and number of exemptions. The standard deduction ($10,000) and standard exemption ($4,150) are always used and the tax rate is assumed to be 20%. Return the estimated tax.

>>> et(20000, 1)  #work an example or two offline
CIS 210 Computer Science I

Generate an estimate for federal income tax based on reported income and number of exemptions. The standard deduction ($10,000) and standard exemption ($4,150) are always used and the tax rate is assumed to be 20%. Return the estimated tax.

>>> et(20000, 1)  #work an example or two offline

Given (constants): 10,000 std deduct, 4150 exemption, 20% tax rate
Given (arguments): 20,000 gross income and 1 exemption
20,000 – 10,000 = 10,000  subtract standard deduction from income
4,150 * 1 = 4,150  determine total exemptions
10,000 – 4,150 = 5,850  subtract total exemptions
5,850  taxable income
5,850 * .2 = $1,170  multiply by tax rate to determine tax

→ it's an algorithm!
Does this code do what it is supposed to do?

```python
def et(i,n):
    sd = 10000
    t = (i-sd-(4150*n))*0.2
    return t

>>> et(20000, 1)
1170.0
```

Yes, but …
def et(i, n):
    """(number, integer) -> ?? #type contract
    #brief description
    ""
    Generates an estimate for federal income tax based on reported income (i) and number of exemptions (n).
    The standard deduction is always used and the tax rate is assumed to be 20%. The estimated tax is returned.

>>> et(20000, 1)       #examples of use
1170.0
>>> et(35000, 2)
3340.0   ""
sd = 10000
t = (i-sd-(4150*n))*.2
return t
Every docstring of every function should include:

- **type contract** provides type of each parameter and the type of the value returned by the function

- **brief description** that mentions each parameter by name

- **side effects** (e.g., print), if any

- **returned value**

- **simple examples of use**

- **calls/called by** (if any, if helpful)
→ reflects thoughtful design
→ contributes to usability/maintainability of code
→ integrated with Python help function
→ automated testing
→ makes code easier to revise, update

“One of the characteristics of a well-written function is the ability to read the code [including documentation] and see the underlying algorithm.”
def et(i, n):
    '''(number, integer) -> float
    #type contract
    #brief description
    Generates an estimate for federal income tax based on reported income (i) and number of exemptions (n).
    The standard deduction is always used and the tax rate is assumed to be 20%. The estimated tax is returned.

    >>> et(20000, 1)  #examples of use
    1170.0
    >>> et(35000, 2)
    3340.0  ''
    sd = 10000
    t = (i-sd-(4150*n))*2
    return t
def est_tax(income, exemptions):
    """(float, int) -> float  EXPECTATION FOR CIS 210!
    Generates an estimate for federal income tax … document "gotchas"
    Note that if the gross income is less than the sum of the
    standard deduction and exemption total, the estimated tax will be
    a refundable (negative) number.
    >>> est_tax(20000, 1)
    1170.0
    """

    # Set values needed to generate estimate short comments to supplement code,
    STD_EXEMPT = 4150    e.g. why - not what
    STD_DEDUCT = 10000   observe conventions (here – constants)
    TAX_RATE = .20

    # Calculate federal tax by adjusting
    # reported income and applying tax rate
    taxable_income = income - STD_DEDUCT code clearly reflects underlying algorithm
    exempt_adjust = STD_EXEMPT * exemptions
    taxable_income = taxable_income - exempt_adjust
    estimated_tax = taxable_income * TAX_RATE

    return estimated_tax  returned value is easy to find
CIS 210 Computer Science I

✓ Does this code do what it is supposed to do?
✓ CIS 210 style guidelines (Programming Best Practices)

Python functions
   -- why functions?
   -- what happens when a function is called?
   -- activation record/call stack
   -- Python parameter passing
   -- functions are small programs
   -- functions always return values
   -- functions may cause side effects
   -- functions can call functions (lab, too)

Python Standard Library (importing modules) (lab, too)

Accumulator pattern
   → Python repeat (for)

How to Start Project 2  Turtle graphics (lab only)
Recall:

Python built-in functions (__builtins__)  

User-defined functions

Also:

More functions in the Python Standard Library
Why functions?

Functions : programs :: paragraphs : essays

“fix it and forget it” - support abstraction
Why functions? Functions contribute to:

- program organization (abstraction)
- program readability
  
  **AND**
  
  - program correctness
  - code re-use
Built-in functions (__builtins__)  

Python Standard Library (must be imported)

User-defined functions – Recall:

Defining a function is like defining a variable.

The function name refers to the function value (the body of the function).
Recall: functions are an executable data type.

```python
def twice(x):    # defining a function
    '''    # x is a parameter
    result = x * 2    # function code
    return result    # specify returned value
```
Recall: functions are an **executable** data type.

```python
def twice(x):  # defining a function
    # x is a parameter
    result = x * 2  # function code
    return result  # specify returned value
```

```python
>>> twice
<function twice at 0x104e7b510>
```
Recall: functions are an executable data type.

def twice(x):  # defining a function
    # x is a parameter
    result = x * 2  # function code
    return result  # specify returned value

They must be called to execute (run):

>>> twice(3)  # 3 is an argument
6  # func call is an expression  # evaluates to a value
When a function is called/executed, Python:

1. evaluates each argument one at a time, working from left to right

2. assigns the resulting values to the function parameters

3. creates a space (*activation record*) on the *call stack* to keep track of function execution – return address and local variables (*local namespace*)

4. executes the function until return statement
When a function is called/executed, Python:

1. evaluates each argument one at a time, working from left to right
2. assigns the resulting values to the function parameters
3. creates a space (activation record) on the call stack to keep track of function execution – return address and local variables (local namespace)
4. executes the function until return statement
5. stops function execution and returns value specified in return statement
6. the activation record is (eventually) discarded
7. processing resumes where the function was called
def twice(x):  x is a parameter
        """ (int) -> int
        """
        result = x * 2
        return result

>>> twice(3)  3 is an argument
6
>>> twice(5)  5 is an argument
??
parameters (formal parameters) are variable names supplied when the function is defined.

arguments (actual parameters) are the values supplied when the function is called.

Python – “call by assignment” parameter passing:
parameter name = argument value when the function is called.
Visualize this: When a function is called/executed, Python:

✓ 1. evaluates each argument one at a time, working from left to right: \textbf{5 evaluates to 5}

✓ 2. assigns the resulting values to the function parameters: \texttt{x = 5}

3. creates a space (activation record) to keep track of function execution – return address and local variables (local namespace)

4. executes the function until return statement

5. stops function execution and returns value specified in return statement

6. the activation record is (eventually) discarded

7. processing resumes where the function was called
def twice(x):
    ''' (int) -> int
    '''
    result = x * 2
    return result

>>> twice(3)

x → 5

local namespace
def twice(x):
    ''' (int) -> int
    '''
    result = x * 2
    return result

>>> twice(3)

x → 5 local
result → 10 namespace
def twice(x):
    """ (int) -> int
    """
    result = x * 2
    return result

>>> twice(3)

x → 5 local
result → 10 namespace
  [return 10; resume execution in Shell]
def twice(x):
    ''' (int) -> int
    '''
    result = x * 2
    return result

>>> twice(5)
10 #value returned by executing
   #function is returned to the
   #Shell, where it is printed
Visualize this: When a function is called/executed, Python:

✓ 1. evaluates each argument one at a time, working from left to right: 5 evaluates to 5

✓ 2. assigns the resulting values to the function parameters:
   \( x = 5 \)

✓ 3. creates a space (activation record) to keep track of function execution – return address and local variables (local namespace)

✓ 4. executes the function until return statement

✓ 5. stops function execution and returns value specified in return statement

✓ 6. the activation record is (eventually) discarded

✓ 7. processing resumes where the function was called
def twice(x):
    result = 2 * x
    return result

>>> twice(5)
a) 6  b) 10  c) 99

>>> twice(5)
a) 6  b) 10  c) 99
def atwice(x):
    ...
    result = 2 * x
    return result

def btwice(x):
    ...
    result = 2 * x
    return 99

def ctwice(x):
    ...
    result = 2 * x
    return None

def dtwice(x):
    ...
    result = 2 * x
    print(result)
    return None

>>> twice(5)
??
Functions ALWAYS return a value

(sometimes the value is None)
Functions ALWAYS return a value

(sometimes the value is None)

Functions SOMETIMES cause a side effect
Functions ALWAYS return a value

(sometimes the value is None)

Functions SOMETIMES cause a side effect

side effect: a change, besides the returned value, that persists after the function has finished executing

for example, something is printed
def ctwice(x):
    result = 2 * x
    return result

def dtwice(x):
    result = 2 * x
    print(result)
    return None

def cthrice(x):
    result = ctwice(x) + x
    return result

def dthrice(x):
    result = dtwice(x) + x
    return result

>>> cthrice(5)
??

>>> dthrice(5)
??
CIS 210 Computer Science I

✔ Does this code do what it is supposed to do?
✔ CIS 210 style guidelines (Programming Best Practices)

Python functions
  †✔ why functions?
  †✔ what happens when a function is called?
  †✔ activation record/call stack
  †✔ Python parameter passing
  †✔ functions are small programs
-- functions always return values
-- functions may cause side effects
-- functions can call functions (lab, too)

Python Standard Library (importing modules) (lab, too)

Accumulator pattern
→ Python repeat (for)

How to Start Project 2 - 2 (and all projects)  Turtle graphics (lab only)
def ctwice(x):
    result = 2 * x
    return result

def cthree(x):
    result = ctwice(x) + x
    return result

>>> cthree(5)
??
def ztwice(x):
    """
    result = 2 * x
    return result

def zthrice(x):
    """
    ztwice(x) + x
    return result

>>> zthrice(5)
??
def twice(x):
    result = x * 2
    return result

>>> twice(4)    editor/Run Module: twice(4)
??            ??
def twice(x):
    result = x * 2
    print(result)
    return None

>>> twice(4)

```python
>>> twice(4)  # editor/Run Module: twice(4)
??          # ??
```
def ctwice(x):
    result = 2 * x
    return result

def dtwice(x):
    result = 2 * x
    print(result)
    return None

def cthrice(x):
    result = ctwice(x) + x
    return result

def dthrice(x):
    result = dtwice(x) + x
    return result

>>> cthrice(5)
15

>>> dthrice(5)
??
Functions ALWAYS return a value

(sometimes the value is None)

Functions SOMETIMES cause a side effect

side effect: a change, besides the returned value, that persists after the function has finished executing

for example, something is printed

BE CLEAR ABOUT WHAT THE FUNCTION SHOULD DO: RETURN A VALUE? PRINT A VALUE? (SEE PROJECT SPECIFICATION)
def twice(x):
    
    x = 3
    result = x * 2
    return result

>>> twice(5) 
a) 6
??
b) 10

>>> twice()
c) error
??
d) <function twice at 0x100560e18>

>>> twice
??
A. `def twice(x):`
   ```python
   result = 2 * x
   return result
   ```

B. `def twice(x):`
   ```python
   result = 2 * x
   return None
   ```

C. `def twice(x):`
   ```python
   result = 2 * x
   return 99
   ```

D. `def twice(x):`
   ```python
   result = 2 * x
   print(result)
   return None
   ```

def doubleDouble(x):
   ```python
   result = twice(x) + twice(x)
   return result
   ```

```python
>>> doubleDouble(4)
```
CIS 210 Computer Science I

✓ Does this code do what it is supposed to do?
✓ CIS 210 style guidelines (Programming Best Practices)

Python functions
✓ why functions?
†✓ what happens when a function is called?
✓ activation record/call stack
✓ Python parameter passing
✓ functions are small programs
✓ functions always return values
✓ functions may cause side effects
✓ functions can call functions (lab, too)

Python Standard Library (importing modules) (lab, too)

Accumulator pattern
→ Python repeat (for)

How to Start Project 2  Turtle graphics (lab)
Does this code do what it is supposed to do?
✓ CIS 210 style guidelines (Programming Best Practices)

Python functions
✓ why functions?
†✓ what happens when a function is called?
✓ activation record/call stack
✓ Python parameter passing
✓ functions are small programs
✓ functions always return values
✓ functions may cause side effects
✓ functions can call functions (lab, too)

Python Standard Library (importing modules) (lab, too)

Accumulator pattern
→ Python repeat (for)

How to Start Project 2  Turtle graphics (lab)
Recall:

Python organizes and keeps track of identifiers in namespaces.

When Python starts up, two namespaces are available:

__builtins__ - Python built-in objects

__main__ - objects defined during Python session
>>> my_identifier = 4

>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__', 'my_identifier']

>>> __name__
'__main__'
>>> dir(__builtins__)
[..., 'False', ..., 'NameError', 'None', ..., 'True',
'TypeError', ..., 'abs', 'all', 'any', 'ascii', 'bin', 'bool',
'bytearray', 'bytes', 'callable', 'chr', 'classmethod',
'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict',
dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit', 'filter',
'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr',
'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance',
'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'map',
'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open',
'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr',
'reversed', 'round', 'set', 'setattr', 'slice', 'sorted',
'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars',
'zip']
how can we extend the language – create our own elements?
– user-defined functions
– user-defined data types (CIS 211)

Recall: Python functions name an operation, e.g., built-in function abs

```python
>>> abs
>>> help(abs)
<built-in function abs> ... 
>>> abs() # functions are callable
??
```
More Python functions and values are available in modules (.py files) in the Python Standard Library.

Accessing the Python Standard Library

```python
import math
dir()

from math import pi
dir()

from math import *
dir()
```
Recall: at Python startup – two namespaces:

```python
__builtins__
__main__ (global)
```

```python
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__']
```

```python
>>> __name__
'__main__'
```
>>> x = 99

>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__', 'x']

>>> import math

>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__', 'math', 'x']
>>> x = 99

>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__', 'x']

>>> import math

>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__', 'math', 'x']

A reference to the imported module is added to the __main__ namespace.
>>> x = 99
>>> import math
>>> dir()
['__annotations__', '__builtins__', '__doc__',
'__loader__', '__name__', '__package__', '__spec__',
'math', 'x']

A reference to the imported module is added to the __main__ namespace.

>>> sqrt(81)    >>> math.sqrt(81)
??             ??
>>> from math import pi

>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__', 'math', 'x', 'pi']

A reference to the imported function **function** is added to the **__main__** namespace.

```python
>>> pi
??

>>> math.pi
??
```
>>> from turtle import fd

>>> dir()
['__annotations__', '__builtins__', '__doc__',
'__loader__', '__name__', '__package__', '__spec__', 'fd',
'math', 'x', 'pi']

>>> turtle.bk(50)    >>> turtle.fd(50)    >>> fd(50)
??    ??    ??
>>> from turtle import *

>>> dir()
['__annotations__', '__builtins__', '__doc__',
'__loader__', '__name__', '__package__', '__spec__', 'bk', 'fd', [all turtle functions and values], 'math', 'x', pi]

>>> turtle.bk(50)     >>> bk(50)     >>> fd(50)
??                  ??                  ??
```python
>>> import math
>>> dir(math)
```
Does this code do what it is supposed to do?
CIS 210 style guidelines (Programming Best Practices)

Python functions
  ✓ why functions?
  †✓ what happens when a function is called?
  ✓ activation record/call stack
  ✓ Python parameter passing
  ✓ functions are small programs
  ✓ functions always return values
  ✓ functions may cause side effects
  ✓ functions can call functions (lab, too)

Python Standard Library (importing modules) (lab, too)

Accumulator pattern → Python repeat (for)

How to Start Project 2  Turtle graphics (lab)
accumulator pattern

- initialize accumulator variable
- repeatedly adjust the accumulator variable
- until done
accumulator pattern

x = 1
x = x + 1
x = x + 1
x = x + 1
x

?? DRY
accumulator pattern

\[
\begin{align*}
  x &= 1 \\
  x &= x + 1 \\
  x &= x + 1 \\
  x &= x + 1 \\
  x &= x + 1 \\
\end{align*}
\]

repeat 3 times:

\[
\begin{align*}
  x &= x + 1 \\
\end{align*}
\]
Repeat operation in Python

for <var> in <sequence>:
    <do something>
Repeat operation in Python

for <var> in <sequence>:
  <do something>

for i in range(3):
  print(i)
Repeat operation in Python

for <var> in range <int>:
    <do something>

for item in range(3):
    print(item)

0
1
2
>>> help(range)
Help on class range in module builtins:

class range(object)
 | range(stop) -> range object
 | range(start, stop[, step]) -> range object
 |
 | Return an object that produces a sequence of integers from start (inclusive)
 | to stop (exclusive) by step. range(i, j) produces i, i+1, i+2, ... , j-1.
 | start defaults to 0, and stop is omitted! range(4) produces 0, 1, 2, 3.
 | These are exactly the valid indices for a list of 4 elements.
 | When step is given, it specifies the increment (or decrement).
## accumulator pattern

<table>
<thead>
<tr>
<th>recall:</th>
<th>pseudocode:</th>
<th>Python:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = 1$</td>
<td>$x = 1$</td>
<td>$x = 1$</td>
</tr>
<tr>
<td>$x = x + 1$</td>
<td>repeat 3 times</td>
<td>for $i$ in range(3):</td>
</tr>
<tr>
<td>$x = x + 1$</td>
<td></td>
<td>$x = x + 1$</td>
</tr>
<tr>
<td>$x = x + 1$</td>
<td></td>
<td># $x += 1$</td>
</tr>
</tbody>
</table>
CIS 210 Computer Science I

✓ Does this code do what it is supposed to do?
✓ CIS 210 style guidelines (Programming Best Practices)

Python functions
✓ why functions?
†✓ what happens when a function is called?
✓ activation record/call stack
✓ Python parameter passing
✓ functions are small programs
✓ functions always return values
✓ functions may cause side effects
✓ functions can call functions (lab, too)

✓ Python Standard Library (importing modules) (lab, too)

✓ Accumulator pattern → Python repeat (for)

How to Start Project 2  Turtle graphics (lab)
Project 2-2 – Getting Started

Review and understand project specification – mysqrt, sqrt_compare, main

main calls sqrt_compare calls mysqrt

write main, then the lowest level function (sqrt)
Projects – Getting Started

`mysqrt`

`mysqrt` will have two parameters, `n`, a positive integer to find the square root of, and `k`, the number of times the iterative square root approximation process should run. The function should return the approximate square root value for `n`. 
mysqrt will have two parameters, n, a positive integer to find the square root of, and k, the number of times the iterative square root approximation process should run. The function should return the approximate square root value for n.

mysqrt(4) should give a result around 2
mysqrt(100) → 10-ish  mysqrt(1) → approx. 1
Babylonian Method

\[ x_{k+1} = \frac{1}{2} \cdot \left( x_k + \frac{n}{x_k} \right), \text{ where } x_0 = 1 \]

Develop a thorough understanding of the algorithm
Develop thorough understanding of the algorithm:

\[ n \cdot x \cdot k \]

\[ x_{k+1} = \frac{1}{2} \cdot \left( x_k + \frac{n}{x_k} \right), \text{ where } x_0 = 1 \]
Projects – Getting Started

**mysqrt** will have two parameters, **n**, a positive integer to find the square root of, and **k**, the number of times the iterative square root approximation process should run. The function should **return** the approximate square root value for **n**.
Develop thorough understanding of the algorithm:

\[ n \? x \? k \? \ldots \ x \text{ is an approximate square root of } n, \]

initially set to 1. \( k \) is number of loop iterations.

successive values of \( x \) are better approximations.

\[ x_{k+1} = \frac{1}{2} \cdot \left( x_k + \frac{n}{x_k} \right), \text{ where } x_0 = 1 \]
Develop thorough understanding of the algorithm:

work out an example or two by hand ...

\[ x_{k+1} = \frac{1}{2} \cdot \left( x_k + \frac{n}{x_k} \right), \text{ where } x_0 = 1 \]
for \( n = 4 \) and \( k = 3 \)

\[
\begin{align*}
\text{for } n = 4 \text{ and } k = 3 \\
x_0 &= 1 \\
x_1 &= .5 \cdot (x_0 + n/x_0) = .5 \cdot (1 + 4/1) = 2.5 \\
x_2 &= .5 \cdot (x_1 + n/x_1) = .5 \cdot (2.5 + 4/2.5) = 2.05 \\
x_3 &= .5 \cdot (x_2 + n/x_2) = .5 \cdot (2.05 + 4/2.05) = 2.0006
\end{align*}
\]

\[
x_{k+1} = \frac{1}{2} \cdot \left( x_k + \frac{n}{x_k} \right), \text{ where } x_0 = 1
\]
declarative $\rightarrow$ procedural semantics

**Iterative method:** A first approximation is produced, then a method which improves the accuracy of the solution is used for a certain number of iterations or until two successive approximations agree to the accuracy required.

$$x_{k+1} = \frac{1}{2} \cdot \left( x_k + \frac{n}{x_k} \right), \text{ where } x_0 = 1$$
for $n = 4$ and $k = 3$

$x_0 = 1$  
$x_1 = .5 * (x_0 + n/x_0)$  
$x_2 = .5 * (x_1 + n/x_1)$  
$x_3 = .5 * (x_2 + n/x_2)$

$x_0 = 1$  
$x_1 = .5 * (1 + 4/1) = 2.5$  
$x_2 = .5 * (2.5 + 4/2.5) = 2.05$  
$x_3 = .5 * (2.05 + 4/2.05) = 2.0006$

Note: accumulator pattern – initialize, adjust, until done

$x_{k+1} = x_k$, adjusted  
$x_{\text{new}} = x_{\text{prior}}$, adjusted
for $n = 4$ and $k = 3$

$x_0 = 1$

$x_1 = .5 \times (x_0 + n/x_0)$

$x_2 = .5 \times (x_1 + n/x_1)$

$x_3 = .5 \times (x_2 + n/x_2)$

$x_0 = 1$

$x_1 = .5 \times (1 + 4/1) = 2.5$

$x_2 = .5 \times (2.5 + 4/2.5) = 2.05$

$x_3 = .5 \times (2.05 + 4/2.05) = 2.0006$

Note: accumulator pattern – initialize, adjust, until done

$x_{k+1} = x_k$, adjusted

$x_{\text{new}} = x_{\text{prior}}$, adjusted

?? in Python
for \( n = 4 \) and \( k = 3 \)

\[
\begin{align*}
x_0 &= 1 \\
x_1 &= 0.5 \times (x_0 + n/x_0) = 2.5 \\
x_2 &= 0.5 \times (x_1 + n/x_1) = 2.05 \\
x_3 &= 0.5 \times (x_2 + n/x_2) = 2.0006
\end{align*}
\]

Note: accumulator pattern – initialize, adjust, until done

\( x_{k+1} = x_k \), adjusted

\( x_{\text{new}} = x_{\text{prior}} \), adjusted

?? in Python

Recall: \( x = x + 1 \)

lhs \( x \) is new, rhs \( x \) is prior
my_sqrt will have two parameters, \( n \), a positive integer to find the square root of, and \( k \), the number of times the iterative square root approximation process should run. The function should return the approximate square root value for \( n \).

```python
def my_sqrt(n, k):
    # start with header
    """(??, ??) \rightarrow ?? # and docstring"

    Return approximate square root of n. …
```

Add: Examples of use
```
return #approx sqrt value # and return
```
def mysqrt(n, k):
    """ (int, int) -> float  1) type contract

    2) brief description
    Generates an approximate square root for n, a positive integer, via an iterative process that runs k times.

    The approximate square root is returned.

    3) simple examples of use
    >>> mysqrt(1, 1)
    1.0
    >>> mysqrt(4, 1)
    2.5
    >>> mysqrt(4, 3)
    2.006
    """
Translate the algorithm into Python code using tools from the Python toolkit (only).

Test code using your simple examples and/or examples given in the projects specifications.

Revise your code until it works for various examples.
Projects – Getting Started

-- review project specification thoroughly
-- write examples of expected results for specified inputs
-- develop, review, and/or revise a problem-solving approach, using natural language, algorithm, pseudocode (not code)
-- check algorithm using your examples

Starting with the lowest level function -
-- write the function header
-- write the function docstring – type contract
-- write the function docstring – brief description
-- write the function docstring – examples of use (use ones developed earlier)
-- write the return statement

-- using tools from the Python toolkit, start writing the body of the function
-- test often
-- test using examples in the docstring, and then project spec, and then others
Python toolkit

numeric data types (int, float) and operations (e.g., +, %, **, pow, round, abs)
NoneType (None)
[Boolean data types]
[strings]
expressions
print

variables (identifiers)
assignment statement
Python repetition – for, while
user-defined functions; def, parameter list, docstrings, code, return
IDLE interactive development environment
Python introspection – help, dir functions

Python Standard Library – math, turtle modules; import
Computer Science overview: support for Computational problem solving
Computational Problem Solving: designing, implementing, checking, revising algorithms/pgms.

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); abstraction
Python Shell is a REPL (read-evaluate-print loop)
Programming language basics: keywords, primitive elements, identifiers
Python primitive elements: Objects - value/attributes, type
Combining primitive elements: Expressions - expressions evaluate to a value
Naming values: Variables/assignment - assignment statements are not expressions and do not return a value; namespaces – builtins and __main__
Functions are an executable (callable) 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
Iterative algorithms; accumulator pattern