CIS 210
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)

---

**CIS 210 Introduction Computer Science**

Does this code do what it is supposed to do?

```python
def et(i, n):
    sd = 10000
    t = (i - sd - (4150 * n)) * .2
    return t
```

---

... according to ... survey of 1,000 developers and 1,000 C-level execs, on average about half of the developer’s working week is spent on maintenance, such as debugging, modifying code, and fixing bad code.

---

**CIS 210 Introduction Computer Science**

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.

```python
def et(i, n):
    sd = 6500
    t = (i - sd - (4150 * n)) * .2
    return t
```

---

>>> et(20000, 1)

1170.0

Yes, but ...

---

**CIS 210 Introduction Computer Science**

Computational Process

Does this code do what it is supposed to do?

>>> et(20000, 1)

1170.0

Yes, but ...

---

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! (though not quite a computational process)
**CIS 210 Introduction Computer Science**

**ADDING DOCUMENTATION HELPS**

**DOES THIS CODE DO WHAT IT IS SUPPOSED TO DO?**

```python
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)
    1170.0
    >>> et(35000, 2)
    3340.0
    '''
sd = 10000
t = (i - sd - (4150*n))*.2
return t
```

**CIS 210**

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)

**CIS 210**

✓ 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)

**CIS 210**

**MUCH BETTER!**

**EXPECTATION FOR CIS 210!**

def est_tax(income, exemptions):
    '''(number, int) -> float
    #type contract
    #brief description
    Generates an estimate for federal income tax …
    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
    STD_EXEMPT = 4150
    STD_DEDUCT = 10000
    TAX_RATE = .20

    # Calculate federal tax by adjusting
    # reported income and applying tax rate
    taxable_income = income - STD_DEDUCT
    exempt_adjust = STD_EXEMPT * exemptions
    taxable_income = taxable_income - exempt_adjust
    estimated_tax = taxable_income * TAX_RATE
    return estimated_tax

**CIS 210**

→ reflects thoughtful design
→ contributes to usability/maintainability of code
→ integrated with Python help function
→ automated testing

“One of the characteristics of a well-written function is the ability to read the code [including documentation] and see the underlying algorithm.”

**CIS 210**

Recall:

Python built-in functions (__builtins__)

User-defined functions

More functions in the Python Standard Library
Why functions? Functions contribute to:
- program organization (abstraction)
- program readability
- program correctness
- code re-use

Built-in functions (__builtins__)
Python Standard Library (must be imported)
User-defined functions

Defining a function is like defining a variable.
The function name refers to the function value (the body of the function).

Functions are an executable data type.

```python
>>> def twice(x):
    # defining a function
    result = x * 2
    return result
# result is returned
```

They must be called to execute (run):

```python
>>> twice(3)
# 3 is an argument
6 # fn 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
5. stops function execution and returns value specified in return statement
6. the activation record is (eventually) destroyed
7. processing resumes where the function was called

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) destroyed
7. processing resumes where the function was called
FUNCTIONS ARE SMALL PROGRAMS: DEFINITIONS AND EXPRESSIONS
EXECUTED SEQUENTIALLY*

```python
def twice(x):
    '''
    '''
    print(x)
    result = x * 2
    return result

>>> twice(3)
3 is an argument
>>> 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.

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

>>> twice(5)
5 is an argument
```

Visualize this: 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) 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) destroyed
7. processing resumes where the function was called

```python
def twice(x):
    '''
    '''
    result = x * 2
    return result

>>> twice(3)  # 3 is an argument
>>> twice(5)  # 5 is an argument
```
Functions ALWAYS return a value
(sometimes the value is None)

Functions SOMETIMES cause a side effect
(a change that persists after the function finishes – for example, something is printed)
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:

```
__builtins__
__main__   (global)
```

```python
>>> 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 is added to the __main__ namespace.

>>> 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__', [all turtle functions and values], 'math', 'x', pi]

>>> turtle.bk(50) >>> bk(50) >>> fd(50)
?? ?? ??

accumulator pattern

- initialize accumulator variable
- repeatedly adjust the accumulator variable
- until done
CIS 210

accumulator pattern

\[
x = 1 \\
x = x + 1 \\
x = x + 1 \\
x = x + 1 \\
x
??
\]

CIS 210

Repeat operation in Python

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

CIS 210

Repeat operation in Python

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

for item in range(4):
print(item)

0
1
2
3
Repeat operation in Python

```python
for item in range(4):
    print(item)
```

Python Shell:

```python
>>> range(4)  # returns a range object
range(0, 4)
>>> list(range(4))  # that is a sequence
[0, 1, 2, 3]
```

**accumulator pattern**

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

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

```
x += 1
```

---

**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

---

**Projects – Getting Started**

- `mysqrt`

  `mysqrt` will have two parameters, `n`, the number to find the square root for, 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:

\[ X_{k+1} = \frac{1}{2} \times \left( X_k + \frac{n}{X_k} \right), \text{where } X_0 = 1 \]

... \( x \) is an approximate square root of \( n \), initially set to 1. Successive values of \( x \) are better approximations. \( k \) is number of loop iterations.

\[ X_{k+1} = \frac{1}{2} \times \left( X_k + \frac{n}{X_k} \right), \text{where } X_0 = 1 \]

for \( n = 4 \) and \( k = 3 \)

\[
\begin{align*}
X_0 &= 1 \\
X_1 &= 0.5 \times (X_0 + n/X_0) = 2.5 \\
X_2 &= 0.5 \times (X_1 + n/X_1) = 2.05 \\
X_3 &= 0.5 \times (X_2 + n/X_2) = 2.0006
\end{align*}
\]

\[ X_{k+1} = \frac{1}{2} \times \left( X_k + \frac{n}{X_k} \right), \text{where } X_0 = 1 \]

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

\[ X_{k+1} = \frac{1}{2} \times \left( X_k + \frac{n}{X_k} \right), \text{where } X_0 = 1 \]
CIS 210

for n = 4 and k = 3

\[
\begin{align*}
  x_0 &= 1 \\
  x_1 &= .5 \times (x_0 + n/x_0) \\
  x_2 &= .5 \times (x_1 + n/x_1) \\
  x_3 &= .5 \times (x_2 + n/x_2)
\end{align*}
\]

\[
\begin{align*}
  x_0 &= 1 \\
  x_1 &= .5 \times (1 + 4/1) = 2.5 \\
  x_2 &= .5 \times (2.5 + 4/2.5) = 2.05 \\
  x_3 &= .5 \times (2.05 + 4/2.05) = 2.0006
\end{align*}
\]

Note: accumulator pattern – initialize, adjust, until done

\[
\begin{align*}
  x_{k+1} &= x_k, \text{ adjusted} \\
  x_{\text{new}} &= x_{\text{old}}, \text{ adjusted}
\end{align*}
\]

CIS 210

def mysqrt(n, k):
    """ (integer, integer) -> 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, 2)
    2.05
    ""
    pass
    return x

CIS 210

Write a template for the function – header, docstring, return

What are function parameters? What is the result?
-- number to find square root for -- approximate square root
-- number of times to iterate

How is it reported?
-- value is returned

def mysqrt(n, k):
    ""
    ""
    pass
    return x

CIS 210

Translate the algorithm into Python code using tools from the Python toolkit.

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

Debug and revise your code until it works.
CIS 210

Python toolkit so far

numeric data types (int, float) and operations (e.g., +, **, round, abs, etc.)
(Boolean data types)
(strings)
print
expressions
variables/assignment statement
Python repetition – for (while)

user-defined functions: def, parameter list, docstring, function code, return

Python modules – math, turtle