CIS 210 Intro to 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
```

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)

... 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.

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 = 10000
    t = (i-sd-(4150*n))*.2
    return t
```

Given (constants): 10,000 std deduct, 4150 exemption, 20% tax rate
Given (arguments): 20,000 gross income and 1 exemption

>>> 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))*.2
    return t

>>> et(20000, 1)
1170.0
Yes, but ...
```

---

**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)

---

**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**

→ 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."

---

**Adding Documentation Helps**

**Does this code do what it is supposed to do?**

```python
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

SELF-DOCUMENTING CODE
short comments to supplement code, e.g. why, not what
observe conventions (here – constants)
```

---

**CIS 210**

EVEN BETTER!

**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.

```python
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**

SELF-DOCUMENTING CODE
short comments to supplement code, e.g. why, not what
observe conventions (here – constants)

code clearly reflects underlying algorithm

- returned value is easy to find
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)

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
    result = x * 2
    # function code
    return result
    # specify returned value
```

They must be called to execute (run):

```python
>>> 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
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

**PYTHON PARAMETER PASSING**

```python
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: \texttt{5 evaluates to 5}
✓ 2. assigns the resulting values to the function parameters:
  \begin{verbatim}
  x = 5
  \end{verbatim}
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

```python
def twice(x):
    ''' (int) -> int
    x = 5
    result = x * 2
    return result

>>> twice(3)
```

\begin{verbatim}
x \rightarrow 5
result \rightarrow 10
\end{verbatim}

[return 10; resume execution in Shell]
def twice(x):
    result = 2 * x
    return result

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

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
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 2-2 (and all projects) Turtle graphics (lab only)

---

CIS 210

def ctwice(x):
    result = 2 * x
    return result

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

>>> cthrice(5)
??

---

CIS 210 Introduction Computer Science

def ztwice(x):
    result = 2 * x
    return result

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

>>> zthrice(5)
??

---

CIS 210

def twice(x):
    result = x * 2
    return None

>>> twice(4)
editor/Run Module: twice(4)
??

---

CIS 210 Introduction Computer Science

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

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

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

>>> cthrice(5)
15
??

---
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)

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

>>> twice(5)
a) 6
??
b) 10
??
c) error
??
d) <function twice at 0x100560e18>

>>> twice
??
```

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__']

>>> __name__
'__main__'
```

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)
>>> 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.

>>> sqrt(81)
>>>
math.sqrt(81)
??

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

>>> from math import pi
>>> dir()
['__annotations__', '__builtins__', '__doc__',
'_loader__', '_name__', '_package__', '_spec__', 'math', 'x', 'pi']

>>> 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)
??  ??  ??

A reference to all turtle functions and values is added to the __main__ namespace.

>>> 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)
??  ??  ??
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

Repeat operation in Python

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

DRY

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

x = 1
x = x + 1
x = x + 1
for repeat 3 times:
    x = x + 1
    x = x + 1
    x = x + 1

DRY
Repeat operation in Python

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

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

## Accumulator Pattern

**Recall:**

- `x = x + 1` repeat 3 times

**Pseudocode:**

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

**Python:**

```python
for i in range(3):
    x = x + 1
```

# x += 1

---

Does this code do what it is supposed to do?

- ✓ 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)
CIS 210
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(4)` should give a result around 2
`mysqrt(100)` ~ 10-ish  
`mysqrt(1)` ~ approx. 1

---

CIS 210
Projects – Getting Started – Algorithm??

`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

---

CIS 210
Projects – Getting Started

Develop thorough understanding of the algorithm:

\[
X_{k+1} = \frac{1}{2} \times \left( X_k + \frac{n}{X_k} \right), \text{ where } X_0 = 1
\]
Develop thorough understanding of the algorithm:

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

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

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

\[ x_0 = 1 \] \hspace{1cm} \[ x_0 = 1 \] 
\[ x_1 = .5 \times (1 + 4/1) = 2.5 \] 
\[ x_2 = .5 \times (2.5 + 4/2.5) = 2.05 \] 
\[ x_3 = .5 \times (2.05 + 4/2.05) = 2.0006 \]

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

for n = 4 and k = 3

\[
x_0 = 1 \\
x_1 = 0.5 \times (x_0 + n/x_0) = 0.5 \times (1 + 4/1) = 2.5 \\
x_2 = 0.5 \times (x_1 + n/x_1) = 0.5 \times (2.5 + 4/2.5) = 2.05 \\
x_3 = 0.5 \times (x_2 + n/x_2) = 0.5 \times (2.05 + 4/2.05) = 2.0006
\]

Note: accumulator pattern – initialize, adjust, until done

\[
x_{k+1} = x_k \text{, adjusted} \\
x_{\text{new}} = x_{\text{prior}} \text{, adjusted}
\]

Recall: \( x = x + 1 \)

def mysqrt(n, k):
    # start with header
    """(?, ?) \to ?? # and docstring"
    
    Return approximate square root of n. ...

    Add: Examples of use
    """
    return #approx sqrt value # and return

CIS 210

Projects – Getting Started – Coding

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.

def mysqrt(n, k):
    """(int, int) \to float 1) type contract  
    """
    """brief description 2)  
    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)  
    2.0  
    >> mysqrt(4, 3)  
    2.006  
    """
### 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

### Programming/Computer Science concepts

- Computer Science overview: support for/Computational problem solving
- 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); 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