CIS 210

Days of coding can save hours of planning.

CIS 210

Does this code do what it is supposed to do?

à 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

à Python repeat (for)

How to Start Project 2

Turtle graphics (lab only)

CIS 210

... 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 Intro to Computer Science

Class Encore:

Does this code do what it is supposed to do?

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

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.

def et(i,n):
    sd = 10000
    t = (i-sd-(4150*n))*.2
    return t
### ADDING DOCUMENTATION HELPS

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

```python
def et(i, n):
    """(number, integer) -> float
    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.
    """
    sd = 10000  # assign sd to 10000
    t = (i - sd - (4150 * n)) * .2  # return statement
    return t
```

**EXAMPLES**

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

### computational process

- It's an algorithm! (though not quite a computational process)

### computational process

#### Code:

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

#### Examples:

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

### EXPECTATION FOR CIS 210!

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

```python
def est_tax(income, exemptions):
    """(number, int) -> float
    Generates an estimate for federal income tax based on reported income and number of exemptions.
    The standard deduction is always used and the tax rate is assumed to be 20%. The estimated tax is returned.
    """
    # 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
```

**EXAMPLES**

```python
>>> est_tax(20000, 1)
1170.0
```

#### Code:

```python
def est_tax(income, exemptions):
    STD_EXEMPT = 4150
    STD_DEDUCT = 10000
    TAX_RATE = .20
    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

**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)

> 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

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)

CIS 210

Recall:

Python built-in functions (__builtins__)
User-defined functions

More functions in the Python Standard Library

CIS 210

Why functions? Functions contribute to:
– program organization (abstraction)
– program readability

AND
– program correctness
– code re-use

CIS 210

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).

CIS 210

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
Functions are an executable data type.

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

>>> twice
<function twice at 0x104e7b510>
```

**FUNCTIONS ARE SMALL PROGRAMS:**
**DEFINITIONS AND EXPRESSIONS EXECUTED SEQUENTIALLY**

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

>>> twice(3)  # function calls are expressions so they evaluate to a value
6
```

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
FUNCTIONS ARE SMALL PROGRAMS:
DEFINITIONS AND EXPRESSIONS EXECUTED SEQUENTIALLY*
def twice(x):
    ...  
    result = x * 2
    return result

>>> twice(3)  # function calls are expressions so they evaluate to a value
6
```
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: \(5\) 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

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

```python
>>> twice(5)
99
```

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 twice(x):
    '''
    result = 2 * x
    return result

>>> twice(5)
??

>>> twice
??

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

>>> doubleDouble(4)
??

✓ 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)

More Python functions and values are available in modules (.py files) in the Python Standard Library
Accessing the Python Standard Library

import math
dir()

from math import pi
dir()

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

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

>>> __name__
'__main__'
```

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

```python
>>> import math
```

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

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

```python
>>> from math import pi
```

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

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

```python
>>> pi
```

```python
>>> math.pi
```

```python
>>> from turtle import fd
```

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

```python
>>> turtle.bk(50)
```

```python
>>> turtle.fd(50)
```

```python
>>> fd(50)
```
```python
>>> from turtle import *
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__', 'bk', 'fd', 'math', 'x', 'pi']

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

accumulator pattern

• initialize accumulator variable

• repeatedly adjust the accumulator variable

• until done

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)

```
>>> 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>
</tbody>
</table>
| x = x + 1       | repeat 3 times        | for i in range(3):
|                 | x = 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
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.

def mysqrt(n, k):
    '''(int, ??) → ??
    Return approximate square root of n. ...
    Add: Examples of use
    '''
    return #approx sqrt value

Develop thorough understanding of the algorithm:

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

n? x? k?

... 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 $k = 3$

$x_0 = 1$
$x_1 = .5 \times \left(x_0 + \frac{n}{x_0}\right)$
$x_2 = .5 \times \left(x_1 + \frac{n}{x_1}\right)$
$x_3 = .5 \times \left(x_2 + \frac{n}{x_2}\right)$

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

Note: accumulator pattern – initialize, adjust, until done
$x_{new} = x_{old}$, adjusted

CIS 210

for $n = 4$ and $k = 3$

$x_0 = 1$
$x_1 = .5 \times \left(x_0 + \frac{n}{x_0}\right)$
$x_2 = .5 \times \left(x_1 + \frac{n}{x_1}\right)$
$x_3 = .5 \times \left(x_2 + \frac{n}{x_2}\right)$

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

Note: accumulator pattern – initialize, adjust, until done
$x_{new} = x_{old}$, adjusted

Recall: $x = x + 1$

CIS 210

def mysqrt(n, k):
    '''
    (int, ??) -> ??
    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.
    '''
    x = n
    for _ in range(k):
        x = .5 * (x + n/x)
    return x

>>> mysqrt(1, 1)
1.0
>>> mysqrt(4, 1)
2.5
>>> mysqrt(4, 3)
2.006

CIS 210

declarative $\rightarrow$ procedural semantics

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.

$x_{k+1} = \frac{1}{2} \times \left(x_k + \frac{n}{x_k}\right), \text{where } x_0 = 1$
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.

Programming/Computer Science concepts

1/17/19