CS 61A Midterm 1 Review

Dan Wang, Brandon Wang, and Jonathan Lin

Eta Kappa Nu, Mu Chapter
University of California, Berkeley

12 February 2012

Warmup

Assign a truth value to each of the following statements:

1. Every function in Python is a pure function.

2. Every variable is an object in Python.

3. The following code will cause an error:

```
1 a = 1
2 print(a.__add__(2))
```

Warmup

Assign a truth value to each of the following statements:

1. Every function in Python is a pure function.
   False.
2. Every variable is an object in Python.

3. The following code will cause an error:

```
1 a = 1
2 print(a.__add__(2))
```

Warmup

Assign a truth value to each of the following statements:

1. Every function in Python is a pure function.
   False.
2. Every variable is an object in Python.
   True.
3. The following code will cause an error:

```
1 a = 1
2 print(a.__add__(2))
```

Warmup

Assign a truth value to each of the following statements:

1. Every function in Python is a pure function.
   False.
2. Every variable is an object in Python.
   True.
3. The following code will cause an error:

```
1 a = 1
2 print(a.__add__(2))
```

   False.

What will Python print?

```
>>> print(3)
```

```
>>> x = lambda x: lambda: x * 3
>>> x
```

```
>>> x(3)
```

```
>>> x(3)(4)
```

```
>>> t = (1, (2, 3), (4, (5), (6, 7)))
>>> t[1][0]
```

```
>>> t[2][1][0]
```

What will Python print?

```
>>> print(3)
3
>>> x = lambda x: lambda: x * 3
>>> x
```

```
>>> x(3)
```

```
>>> x(3)(4)
```

```
>>> t = (1, (2, 3), (4, (5), (6, 7)))
>>> t[1][0]
```

```
>>> t[2][1][0]
```

What will Python print?

```
>>> print(3)
3
>>> x = lambda x: lambda: x * 3
>>> x
<function <lambda> at 0x...>
>>> x(3)


>>> x(3)(4)


>>> t = (1, (2, 3), (4, (5), (6, 7)))
>>> t[1][0]


>>> t[2][1][0]
```

What will Python print?

```
>>> print(3)
3
>>> x = lambda x: lambda: x * 3
>>> x
<function <lambda> at 0x...>
>>> x(3)
<function <lambda> at 0x...>
>>> x(3)(4)


>>> t = (1, (2, 3), (4, (5), (6, 7)))
>>> t[1][0]


>>> t[2][1][0]
```

What will Python print?

```
>>> print(3)
3
>>> x = lambda x: lambda: x * 3
>>> x
<function <lambda> at 0x...>
>>> x(3)
<function <lambda> at 0x...>
>>> x(3)(4)
Error
>>> t = (1, (2, 3), (4, (5), (6, 7)))
>>> t[1][0]


>>> t[2][1][0]
```

What will Python print?

```
>>> print(3)
3
>>> x = lambda x: lambda: x * 3
>>> x
<function <lambda> at 0x...>
>>> x(3)
<function <lambda> at 0x...>
>>> x(3)(4)
Error
>>> t = (1, (2, 3), (4, (5), (6, 7)))
>>> t[1][0]
2
>>> t[2][1][0]
Error
```

Functions

Consider the following piece of code:

```
1  def a(m):
2      m = 2
3      print(m)
4      return m
5
6  def b(m):
7      m = m + 6
8      return m
9
10 def c(m):
11     return a(b(m))
12
13 def d(m):
14     return a(b(d(m)))
15
16 q = 10
```

Which of these functions is pure?

What will be the result of `c(c(q))`?

What will be the result of `d(q)`?

What will be the value of `q` after calling `b(q)`?

Functions

Consider the following piece of code:

```
1  def a(m):
2      m = 2
3      print(m)
4      return m
5
6  def b(m):
7      m = m + 6
8      return m
9
10 def c(m):
11     return a(b(m))
12
13 def d(m):
14     return a(b(d(m)))
15
16 q = 10
```

Which of these functions is pure?
`b`

What will be the result of `c(c(q))`?

What will be the result of `d(q)`?

What will be the value of `q` after calling `b(q)`?

Functions

Consider the following piece of code:

```
1  def a(m):
2      m = 2
3      print(m)
4      return m
5
6  def b(m):
7      m = m + 6
8      return m
9
10 def c(m):
11     return a(b(m))
12
13 def d(m):
14     return a(b(d(m)))
15
16 q = 10
```

Which of these functions is pure?
b

What will be the result of `c(c(q))`?
2

What will be the result of `d(q)`?

What will be the value of `q` after calling `b(q)`?

Functions

Consider the following piece of code:

```
1  def a(m):
2      m = 2
3      print(m)
4      return m
5
6  def b(m):
7      m = m + 6
8      return m
9
10 def c(m):
11     return a(b(m))
12
13 def d(m):
14     return a(b(d(m)))
15
16 q = 10
```

Which of these functions is pure?

b

What will be the result of `c(c(q))`?

2

What will be the result of `d(q)`?

`Infinite Loop`

What will be the value of `q` after calling `b(q)`?

Functions

Consider the following piece of code:

```
1  def a(m):
2      m = 2
3      print(m)
4      return m
5
6  def b(m):
7      m = m + 6
8      return m
9
10 def c(m):
11     return a(b(m))
12
13 def d(m):
14     return a(b(d(m)))
15
16 q = 10
```

Which of these functions is pure?
b

What will be the result of `c(c(q))`?
2

What will be the result of `d(q)`?
`Infinite Loop`

What will be the value of `q` after calling `b(q)`?
10

Functions

Consider the function `f`:

```
1 def f():
2     print(12345)
3     return f
```

What will `f()()()` print?

Functions

Consider the function `f`:

```
1 def f():
2     print(12345)
3     return f
```

What will `f()()()` print?

12345
12345
12345

Control Flow

Consider the following function:

```
1 def f(m):
2     while m <= 10:
3         m = m + m + 2
4     return m
```

What will `f(0)` return?
What will `f(1)` return?
What will `f(11)` return?
What will `f(-1)` return?

Control Flow

Consider the following function:

```
def f(m):
    while m <= 10:
        m = m + m + 2
    return m
```

What will `f(0)` return? `14`
What will `f(1)` return?
What will `f(11)` return?
What will `f(-1)` return?

Control Flow

Consider the following function:

```
1 def f(m):
2     while m <= 10:
3         m = m + m + 2
4     return m
```

What will `f(0)` return? 14
What will `f(1)` return? 22
What will `f(11)` return?
What will `f(-1)` return?

Control Flow

Consider the following function:

```
1 def f(m):
2     while m <= 10:
3         m = m + m + 2
4     return m
```

What will `f(0)` return? 14
What will `f(1)` return? 22
What will `f(11)` return? 11
What will `f(-1)` return?

Control Flow

Consider the following function:

```
1 def f(m):
2     while m <= 10:
3         m = m + m + 2
4     return m
```

What will `f(0)` return? 14
What will `f(1)` return? 22
What will `f(11)` return? 11
What will `f(-1)` return? 14

Bizz Buzz

Write a function `bizz_buzz` that takes an integer `n` as an argument. It will then print a line for each number between `1` and `n`, following these rules:

- If the number is divisible by 2, print `"bizz"`
- If the number is divisible by 3, print `"buzz"`
- If the number is divisible by both 2 and 3, print `"bizzbuzz"`
- Otherwise, simply print the number

Here is an example:

```
>>> bizz_buzz(8)
1
bizz
buzz
bizz
5
bizzbuzz
7
bizz
```

Bizz Buzz

Write a function `bizz_buzz` that takes an integer `n` as an argument. It will then print a line for each number between `1` and `n`, following these rules:

- If the number is divisible by 2, print `"bizz"`
- If the number is divisible by 3, print `"buzz"`
- If the number is divisible by both 2 and 3, print `"bizzbuzz"`
- Otherwise, simply print the number

Here is an example:

```
>>> bizz_buzz(8)
1
bizz
buzz
bizz
5
bizzbuzz
7
bizz
```

```
1  def bizz_buzz(n):
2      for i in range(1, n+1):
3          if i % 6 == 0:
4              print('bizzbuzz')
5          elif i % 2 == 0:
6              print('bizz')
7          elif i % 3 == 0:
8              print('buzz')
9          else:
10             print(i)
```

Higher-order functions

- Write a function `commutative` that takes in a function as an argument and returns another function.
- The input function `f` should take in two arguments. Our output function will also take two arguments.
- `commutative` returns a function that returns `True` if the two arguments called could be swapped to have the same return value when called with `f`, and `False` otherwise.
- For example:
  `commutative(add)(1, 2)` should return `True` because `1+2 == 2+1`
- However,
  `commutative(lambda x, y:  x*x+y)(2, 5)` should return `False` because `2*2+5 != 5*5+2`

Higher-order functions

- Write a function `commutative` that takes in a function as an argument and returns another function.
- The input function `f` should take in two arguments. Our output function will also take two arguments.
- `commutative` returns a function that returns `True` if the two arguments called could be swapped to have the same return value when called with `f`, and `False` otherwise.
- For example:
  `commutative(add)(1, 2)` should return `True` because `1+2 == 2+1`
- However,
  `commutative(lambda x, y: x*x+y)(2, 5)` should return `False` because `2*2+5 != 5*5+2`

Solution:

```
1 def commutative(f):
2     return lambda x, y: f(x, y) == f(y, x)
```

Environment Diagrams

```
>>> def foo():
...     x = 1
...     def bar():
...         x = 2
...     def foo():
...         print(x)
...     return foo, bar
...
>>> def hilfinger(me, tom):
...     me()
...     tom()
...
>>> aki, you = foo()
>>> hilfinger(aki, you)


>>> aki()
```

Environment Diagrams

```
>>> def foo():
...     x = 1
...     def bar():
...         x = 2
...     def foo():
...         print(x)
...     return foo, bar
...
>>> def hilfinger(me, tom):
...     me()
...     tom()
...
>>> aki, you = foo()
>>> hilfinger(aki, you)

1

>>> aki()
```

Environment Diagrams

```
>>> def foo():
...     x = 1
...     def bar():
...         x = 2
...     def foo():
...         print(x)
...     return foo, bar
...
>>> def hilfinger(me, tom):
...     me()
...     tom()
...
>>> aki, you = foo()
>>> hilfinger(aki, you)

1

>>> aki()

1
```

RLists

Recall the implementation of `RLists`:

```
1  empty_rlist = None
2
3  def make_rlist(first, rest = empty_rlist):
4      return first, rest
5
6  def first(r):
7      return r[0]
8
9  def rest(r):
10     return r[1]
```

double()

Write a method `double` that takes in an `RList` *r* and returns an `RList` where all elements in *r* appear twice in a row. For example:

```
>>> double(make_rlist(1, make_rlist(2, make_rlist(3))))
(1, (1, (2, (2, (3, (3, None))))))
```

.

double()

Write a method `double` that takes in an `RList` *r* and returns an `RList` where all elements in *r* appear twice in a row. For example:

```
>>> double(make_rlist(1, make_rlist(2, make_rlist(3))))
(1, (1, (2, (2, (3, (3, None))))))
```

.

double()

Write a method `double` that takes in an `RList` *r* and returns an `RList` where all elements in *r* appear twice in a row. For example:

```
>>> double(make_rlist(1, make_rlist(2, make_rlist(3))))
(1, (1, (2, (2, (3, (3, None))))))
```

.

```
def double(r):
    ...
```

double()

Write a method `double` that takes in an `RList` *r* and returns an `RList` where all elements in *r* appear twice in a row. For example:

```
>>> double(make_rlist(1, make_rlist(2, make_rlist(3))))
(1, (1, (2, (2, (3, (3, None))))))
```

.

```
def double(r):
    ...
```

Solution:

```
1 def double(rlist):
2     if rlist == empty_rlist:
3         return rlist
4     else:
5         f = first(rlist)
6         r = rest(rlist)
7         return make_rlist(f, make_rlist(f, double(r)))
```

Abstraction

Now suppose we change our representation of `RLists`:

```
1  empty_rlist = ()
2
3  def make_rlist(first, rest = empty_rlist):
4      return (first, (rest,))
5
6  def first(r):
7      return r[0]
8
9  def rest(r):
10     return r[1][0]
```

Abstraction

Now suppose we change our representation of `RLists`:

```
1  empty_rlist = ()
2
3  def make_rlist(first, rest = empty_rlist):
4      return (first, (rest,))
5
6  def first(r):
7      return r[0]
8
9  def rest(r):
10     return r[1][0]
```

Change `double()` to be compatible with this new `RList` representation.

Abstraction

Now suppose we change our representation of `RLists`:

```
1  empty_rlist = ()
2
3  def make_rlist(first, rest = empty_rlist):
4      return (first, (rest,))
5
6  def first(r):
7      return r[0]
8
9  def rest(r):
10     return r[1][0]
```

Change `double()` to be compatible with this new `RList` representation.
Nothing needs to be changed.