Review

Winter 2019

Major Topics

- Recursion Questions
- Complexity:
  - Linked Lists manipulations
  - Given code, find running time
  - Queues/Stacks
- Queue/Stacks implementations
  - Using arrays (not Python lists)
  - Be able to resize the array
- Linked Lists

Major Topics - 2

- Classes, Inheritance
- repr and str
- Higher - order functions, lambdas
- Data abstraction
- Mutable, Immutable
- Exceptions
- Regex
- Python:
  - Dictionaries, tuples, lists, sets, list comprehension, map, filter

From decimal to binary

Write a function that takes a positive integer and converts it to its binary representation

```
def convert(num):
    """
    >>> convert(1)
    01
    >>> convert(10)
    01010
    >>> convert(20)
    010100
    """
```

Output?

```
s = LinkNode(1, LinkNode(2, LinkNode(3)))
s.value = 5
t = s.next
t.next = s
v = s.next.next.next.next.next.value
print(v)
```

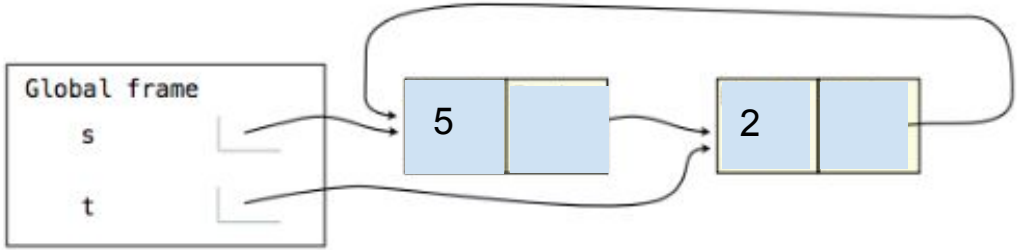
What will be printed?

A: 1
B: 2
C: 3
D: 5
E: Something else

Output?

```
s = LinkNode(1, LinkNode(2, LinkNode(3)))
s.value = 5
t = s.next
t.next = s
print(s.first)  → 5
v = s.next.next.next.next.next.value
print(v)
```

2



Note: The actual environment diagram is much more complicated.

Objects

Instance attributes are found *before* class attributes; class attributes are *inherited*

Objects

Instance attributes are found before class attributes; class attributes are inherited

```
class Worker:
    greeting = 'Sir'
```

Objects (done)

Instance attributes are found before class attributes; class attributes are inherited

```
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
```

Objects

Instance attributes are found before class attributes; class attributes are inherited

```
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
```

Objects

Instance attributes are found before class attributes; class attributes are inherited

```
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting
```

Land Owners

Instance attributes are found before class attributes; class attributes are inherited

```
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
```

Objects

Instance attributes are found before class attributes; class attributes are inherited

```
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
```

Instance attributes are found before class attributes; class attributes are inherited

```
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'
```

```
>>> Worker().work()

>>> jack

>>> jack.work()

>>> john.work()

>>> john.elf.work(john)
```

Objects

```
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'
```

<class Worker>

greeting: 'Sir'

Objects

```
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'
```

<class Worker>

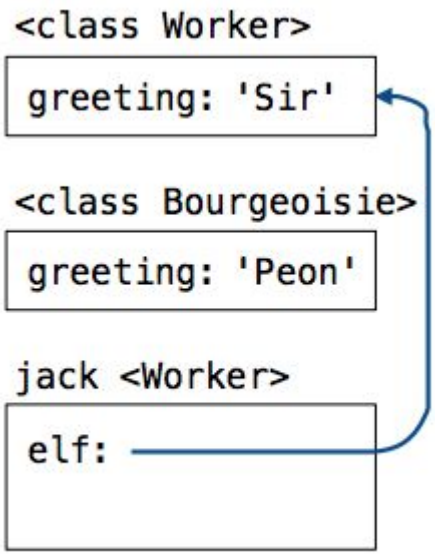
greeting: 'Sir'

<class Bourgeoisie>

greeting: 'Peon'

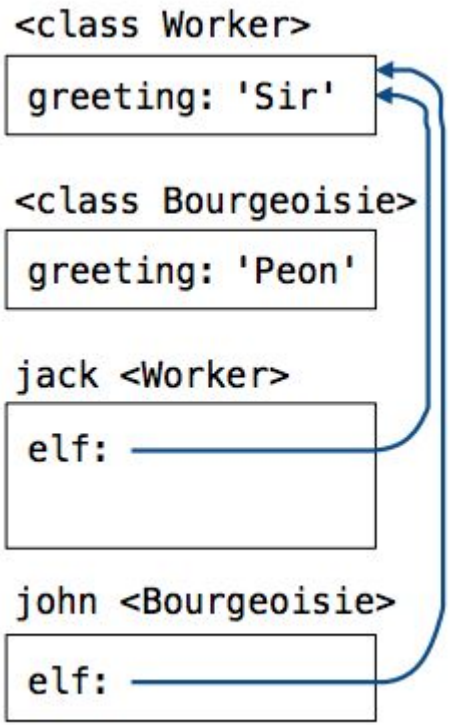
Objects

```
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'
```



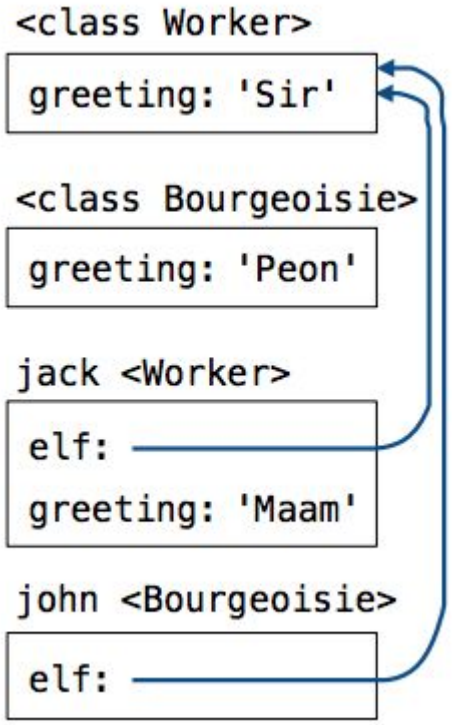
Objects

```
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'
```



Objects

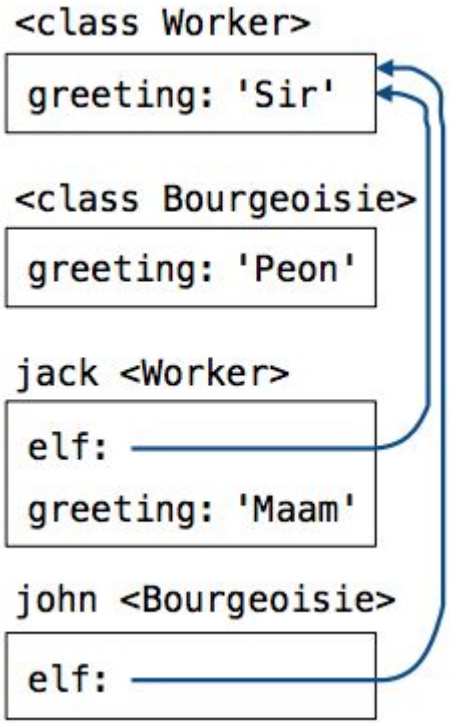
```
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'
```



Objects

```
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'
```

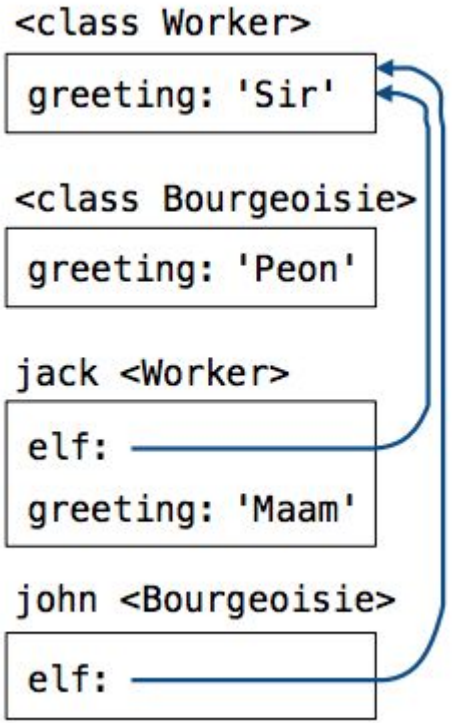
```
>>> Worker().work()
```



Objects

```
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'
```

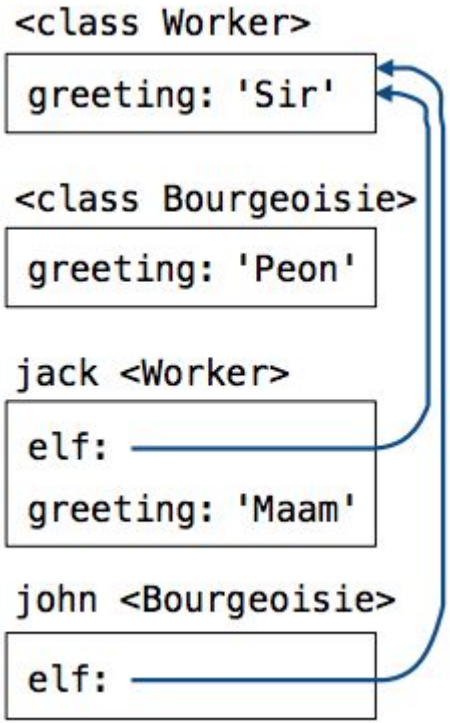
```
>>> Worker().work()
'Sir, I work'
```



Objects

```
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'
```

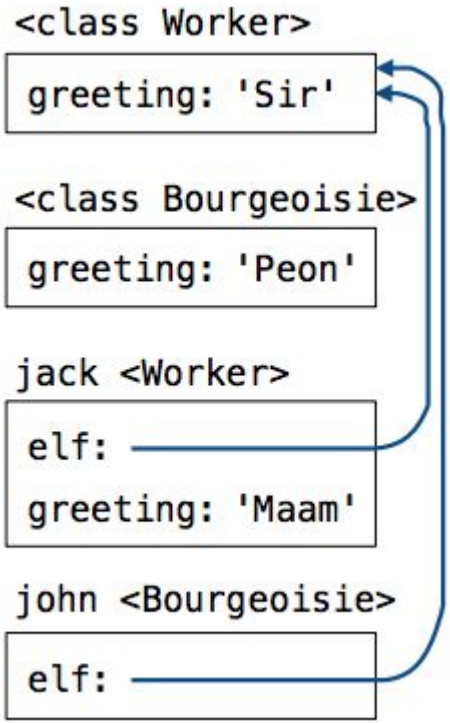
```
>>> Worker().work()
'Sir, I work'

>>> jack
```



Objects

```
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'
```

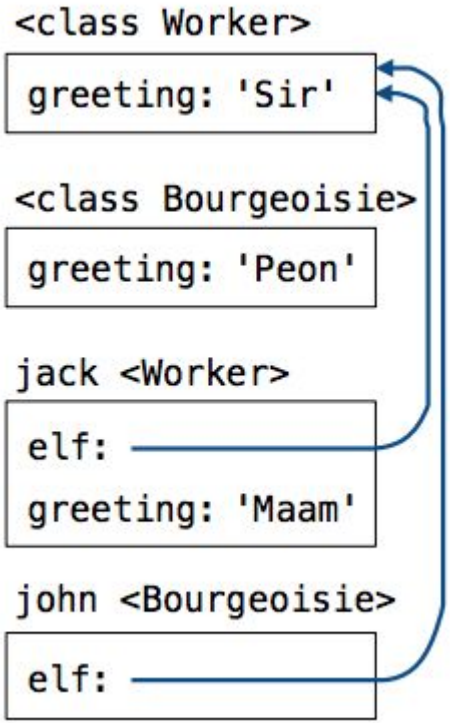
```
>>> Worker().work()
'Sir, I work'

>>> jack
Peon
```



Objects

```
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'
```

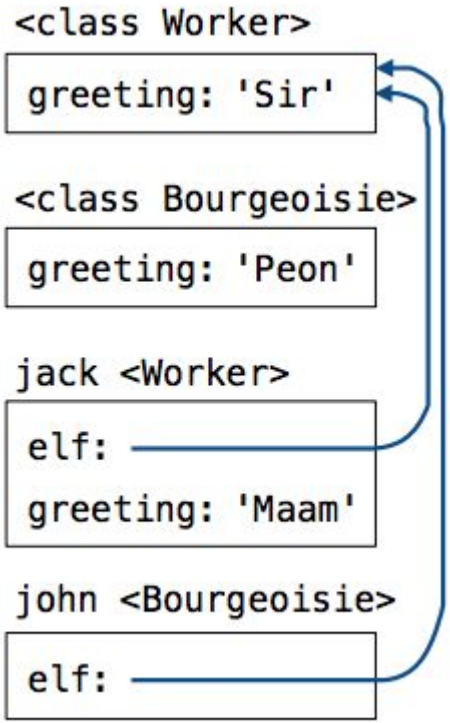
```
>>> Worker().work()
'Sir, I work'

>>> jack
Peon

>>> jack.work()
```



Objects

```
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'
```

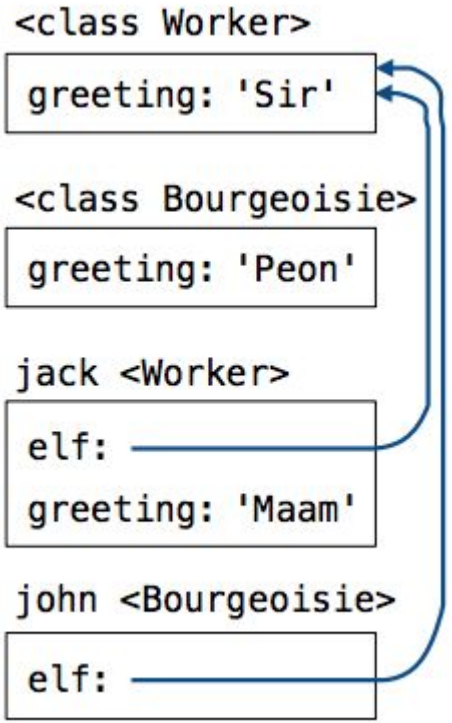
```
>>> Worker().work()
'Sir, I work'

>>> jack
Peon

>>> jack.work()
'Maam, I work'
```



Objects

```
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'
```

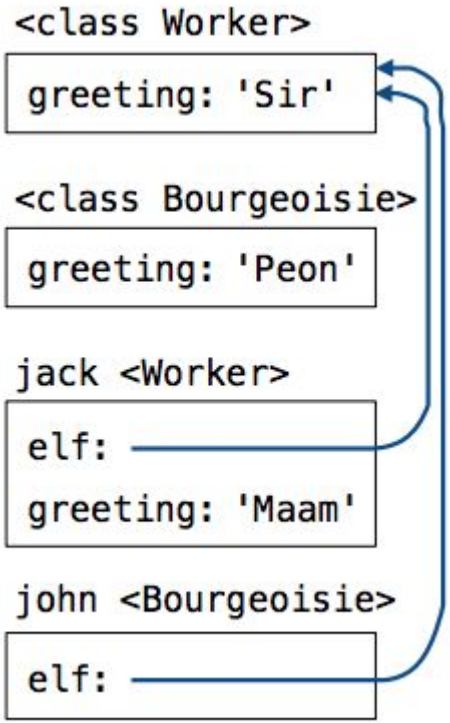
```
>>> Worker().work()
'Sir, I work'

>>> jack
Peon

>>> jack.work()
'Maam, I work'

>>> john.work()
```



Objects

```
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'
```

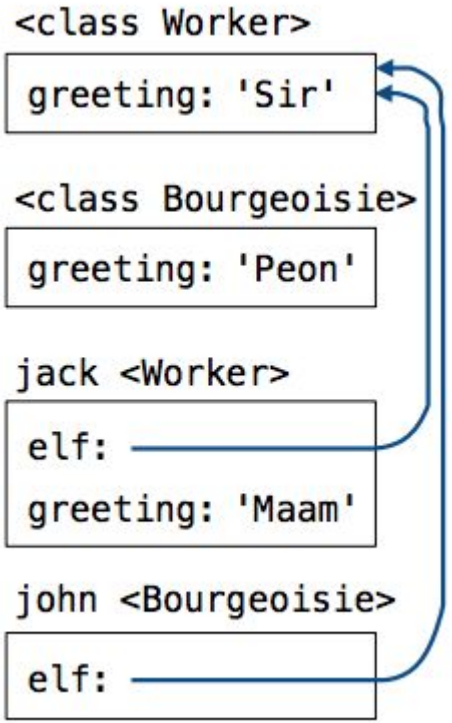
```
>>> Worker().work()
'Sir, I work'

>>> jack
Peon

>>> jack.work()
'Maam, I work'

>>> john.work()
Peon, I work
'I gather wealth'
```



Objects

```
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'
```

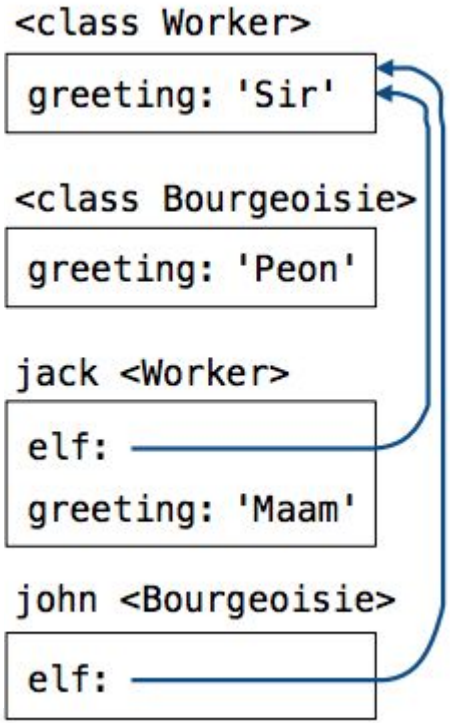
```
>>> Worker().work()
'Sir, I work'

>>> jack
Peon

>>> jack.work()
'Maam, I work'

>>> john.work()
Peon, I work
'I gather wealth'

>>> john.elf.work(john)
```



Objects

```
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'
```

```
>>> Worker().work()
'Sir, I work'

>>> jack
Peon

>>> jack.work()
'Maam, I work'

>>> john.work()
Peon, I work
'I gather wealth'

>>> john.elf.work(john)
'Peon, I work'
```



Announcements

- You are allowed one double sided and handwritten cheat sheet.
- Tuesday, 8am. Same room
- I will create a seating chart for you over the weekend. Check your emails.
- Practice Problems are posted, you need to solve them together
- Review session. Sunday only.

Time

A: Morning <= noon

B: Early afternoon <= 3

C: late afternoon <=6

D: evening <= 9

E: Can’t come

Complexity

```
def func1(n):
    num = 0;
    for i in range(n):
         num = n * i
         j = 1
         while j < n:
           num += i*j
           j = j * 2

    return num

def func2(n):
    num = 0
    for i in range(n):
       num = num + func1(n)
```

Give the running time as a function of `n` if `func2(n)` is called.

Complexity

HW7

Why do we need it?

1) “Simple cases”
2) Given code, convert to “simple cases”

Example

$n^2 + \log n^{10} = \Theta (n^2 + 10)$ -> T/F

What is the complexity and why?

```
def question2(n):
    k = 0
    i = n/2
    j = 2
    for i in range(n):
        print("final")
        for j in range (n):
            k = k + n/2
            print("review")
            j = j*2
        i = i*2
```

From review

1. Linked List operation complexity practice: (Singly linked list.)

    a. What is the complexity to delete the first element in the linked list?
    b. What is the complexity to delete the last element in the linked list?
    c. What is the complexity to add an element at the beginning of linked list?
    d. What is the complexity to add an element at the last of linked list?

repr and str

repr and str

Try to answer:

1) Why do we need functions like str and repr?

2) What is the difference between str and repr

3) Can they give different outputs?

4) Can they give the same outputs?

5) Can repr or str give you an error (given syntax is correct?)

```
>>> x = 'foo'

>>> x

??? - 1

>>> print(x)

??? - 2
```

What are the outputs?

	??? - 1	??? - 2
A:	foo	foo
B:	‘foo’	‘foo’
C:	foo	‘foo’
D:	‘foo’	foo
E:	Something else	

https://stackoverflow.com/questions/7784148/understanding-repr-function-in-python

```
>>> x = 'foo'

>>> x

??? - 1

>>> print(x)

??? - 2
```

What are the outputs?

	??? - 1	??? - 2
A:	foo	foo
B:	‘foo’	‘foo’
C:	foo	‘foo’
D:	**‘foo’**	**foo**
E:	Something else	

https://stackoverflow.com/questions/7784148/understanding-repr-function-in-python

Implementing repr and str

The behavior of **repr** is slightly more complicated than invoking **__repr__** on its argument:

- An **instance** attribute called __repr__ is **ignored**! Only **class** attributes are found

How would we implement this behavior? Which of the following function definitions corresponds to a function **repr** that takes in some argument, looks up the class attribute called **__repr__** and invokes it?

A:
```
def repr(x):
    return type(x).__repr__(x)
```

B:
```
def repr(x):
    return x.__repr__()
```

C:
```
def repr(x):
    return x.__repr__(x)
```

D:
```
def repr(x):
    return type(x).__repr__()
```

E:
```
def repr(x):
    return super(x).__repr__()
```

str

```
def repr(x):
    return type(x).__repr__(x)

def str(x):
    t = type(x)
    if hasattr(t, '__str__'):
        return t.__str__(x)
    else:
        return repr(x)
```

```python
class Test:
    """A Test."""

    def __init__(self):
        self.__repr__ = lambda: 'test'
        self.__str__ = lambda: 'I hate tests'

    def __repr__(self):
        return 'This is Test'

    def __str__(self):
        return 'this test is easy'
```

```python
t = Test()

>>> repr(t)
>>> str(t)
>>> print(t)
>>> t.__repr__()
>>> t.__str__()
```

Stacks and Queues

Stack and Queue running times

For efficient implementation

	Pop/Push	Dequeue/Queue
A:	$\Theta(1)$	$\Theta(n)$
B:	$\Theta(n)$	$\Theta(1)$
C:	$\Theta(1)$	$\Theta(1)$
D:	$\Theta(n)$	$\Theta(n)$
E:	something else	

How can we implement Stacks and Queues

	Stack	Queue
A:	Array only	Array only
B:	Linked List only	Linked List only
C:	Array/LL	Array/LL
D:	LL only	Array/LL
E:	something else	

Queue using an Array

- If you choose to implement a queue using an array, how long does each operation take if the front of the queue is at position 0 in the Array?

- A. add: $\Theta(n)$, remove: $\Theta(n)$, peek: $\Theta(1)$
- B. add: $\Theta(n)$, remove: $\Theta(1)$, peek: $\Theta(1)$
- C. add: $\Theta(\log(n))$, remove: $\Theta(\log(n))$, peek: $\Theta(\log(n))$
- D. add: $\Theta(n)$, remove: $\Theta(n)$, peek: $\Theta(n)$
- E. None of these/other

Stack using an Array

- If you choose to implement a stack using an array how long does each operation take if the top of the stack is at position 0 in the Array?

- A. add: $\Theta(n)$, remove: $\Theta(n)$, peek: $\Theta(1)$
- B. add: $\Theta(n)$, remove: $\Theta(1)$, peek: $\Theta(1)$
- C. add: $\Theta(\log(n))$, remove: $\Theta(\log(n))$, peek: $\Theta(\log(n))$
- D. add: $\Theta(n)$, remove: $\Theta(n)$, peek: $\Theta(n)$
- E. None of these/other

Lambda function

```
>>> j = lambda: lambda x: x*2
```

What is the proper way to call it?

A: `j`

B: `j()`

C: `j(3)`

D: `j()(3)`

E: `j(3)(3)`

Lambda Function

```
>>> l = [lambda: n for n in range(10)]
>>> [f() for f in l]
```

Output?

Dictionaries

Write a function `accept_login(users, username, password)`. The function should return **True** if the user exists and the password is correct and **False** otherwise.

```
users = { "user1" : "password1",
          "user2" : "password2",
          "user3" : "password3"
        }
if accept_login(users, "wronguser", "wrongpassword"):
    print("login successful!")
else :
    print("login failed...")
```

Dictionaries

```
users = { "user1" : "password1",
          "user2" : "password2",
          "user3" : "password3"
        }
if accept_login(users, "wronguser",
"wrongpassword"):
    print("login successful!")
else :
    print("login failed...")
```

```
def accept_login(users, username, password):

    if users.get(username) == None:
        return False
    if users.get(username) != password:
        return False
    else:
        return True
```