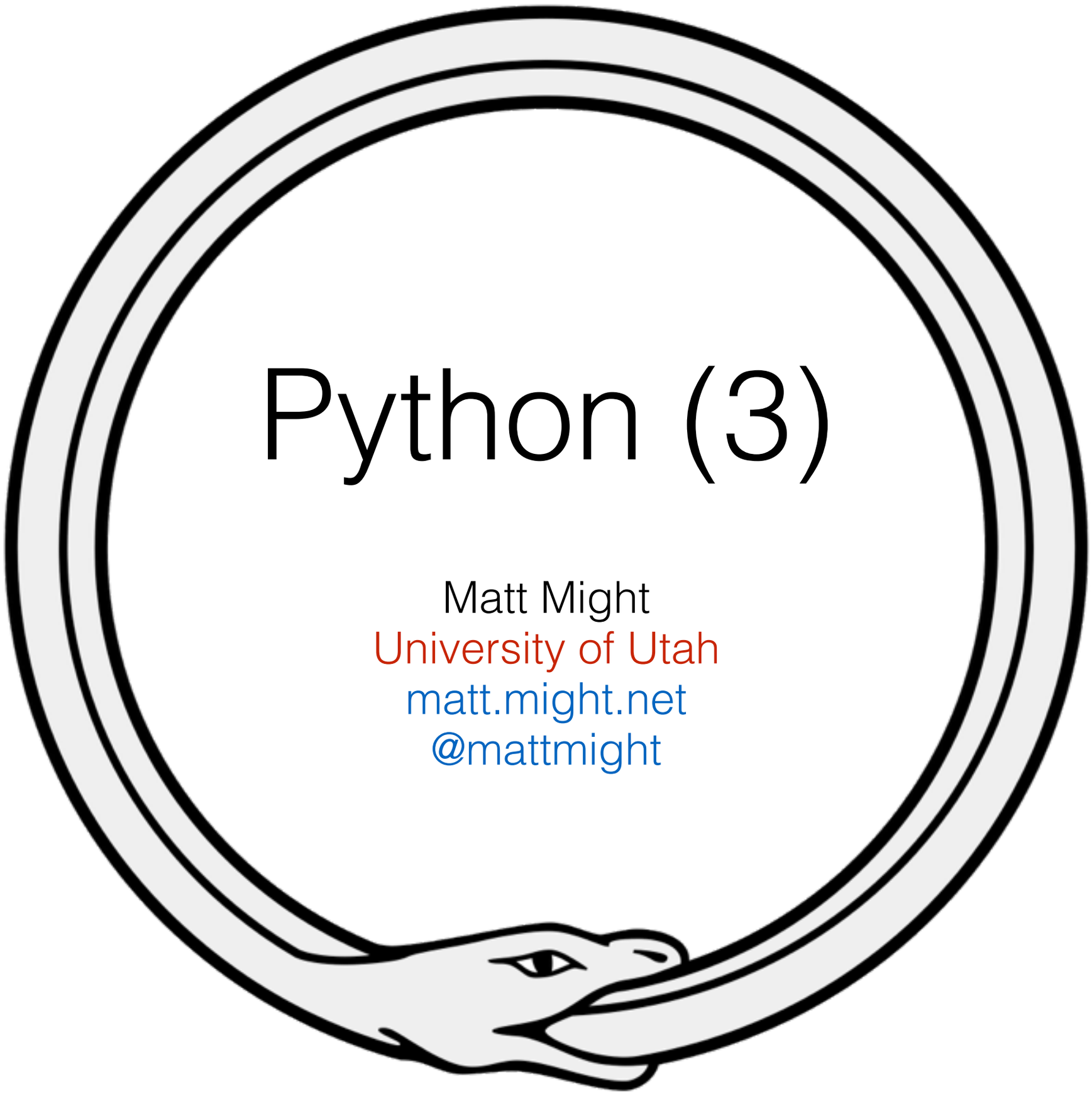
Python (3)

Matt Might
University of Utah
matt.might.net
@mattmight



Simple

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n*fact(n-1)

fact(100)
```

933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000000

Indentation matters

```
def fact(n):
  if n == 0:
return 1
  else:
    return n*fact(n-1)

fact(100)
```

Dynamically typed

Structured

Object-Oriented

Functional*

Programs

Definitions + Statements

Definitions

Variable Definitions

```
x = 10
```

$$\pi = 3.14$$

Function Definitions

```
def f(x):
    x = x + 1
    return x
```

f(20)

Class Definitions

```
class classname:
    statement_1
    ...
    statement_n
```

```
class Animal:
  energy = 0
  def eat(self,calories):
    self.energy += calories

a = Animal()

a.eat(100)

print(a.energy)  # prints 100
```

```
class Animal:
  energy = 0
  def eat(self,calories):
        energy += calories

a = Animal()

a.eat(100) # error
```

Expressions

Atomic Expressions

Keyword expressions

```
... None True False
```

```
False = True
```

(Python 2)

Identifiers

2.3. Identifiers and keywords

Identifiers (also referred to as names) are described by the following lexical definitions.

The syntax of identifiers in Python is based on the Unicode standard annex UAX-31, with elaboration and changes as defined below; see also PEP 3131 for further details.

Within the ASCII range (U+0001..U+007F), the valid characters for identifiers are the same as in Python 2.x: the uppercase and lowercase letters A through Z, the underscore _ and, except for the first character, the digits 0 through 9.

Python 3.0 introduces additional characters from outside the ASCII range (see PEP 3131). For these characters, the classification uses the version of the Unicode Character Database as included in the unicodedata module.

Identifiers are unlimited in length. Case is significant.

```
identifier   ::=  xid_start xid_continue*
id_start     ::=  <all characters in general categories Lu, Ll, Lt, Lm, Lo, Nl, the underscore, and characters with the Other_ID_Start property>
id_continue  ::=  <all characters in id_start, plus characters in the categories Mn, Mc, Nd, Pc and others with the Other_ID_Continue property>
xid_start    ::=  <all characters in id_start whose NFKC normalization is in "id_start xid_continue*">
xid_continue ::=  <all characters in id_continue whose NFKC normalization is in "id_continue*">
```

The Unicode category codes mentioned above stand for:

- Lu - uppercase letters
- Ll - lowercase letters
- Lt - titlecase letters
- Lm - modifier letters
- Lo - other letters
- Nl - letter numbers
- Mn - nonspacing marks
- Mc - spacing combining marks
- Nd - decimal numbers
- Pc - connector punctuations
- Other_ID_Start - explicit list of characters in PropList.txt to support backwards compatibility
- Other_ID_Continue - likewise

All identifiers are converted into the normal form NFKC while parsing; comparison of identifiers is based on NFKC.

A non-normative HTML file listing all valid identifier characters for Unicode 4.1 can be found at http://www.dcl.hpi.uni-potsdam.de/home/loewis/table-3131.html.

foo

π

λ

Λ

λαμδα

Щ

ぼ

㒵

ໜາ

Numbers

0xFF

0o777

0b10101

123.456

```
1j
```

```
1j * 1j == -1
```

Strings

"foo bar"

'foo bar'

'can\'t'

"can't"

```
"He said, \"Hi.\""
```

```
'He said, "Hi."'
```

```
'He said,\n"Hi."'
```

'''He said,
"Hi."'''

"""He said,
"Hi.\""""

b"byte string"

```
"foo \N{GREEK CAPITAL LETTER LAMDA} bar"
```

```
r"foo \N{GREEK CAPITAL LETTER LAMDA} bar"
```

\newline	Backslash and newline ignored
\	Backslash (\)
\'	Single quote (')
\”	Double quote (")
\a	ASCII Bell (BEL)
\b	ASCII Backspace (BS)
\f	ASCII Formfeed (FF)
\n	ASCII Linefeed (LF)
\r	ASCII Carriage Return (CR)
\t	ASCII Horizontal Tab (TAB)
\v	ASCII Vertical Tab (VT)
\ooo	Character with octal value ooo
\xhh	Character with hex value hh

Escape sequences only recognized in string literals are:

\N{name}	Character named name in the Unicode database
\uxxxx	Character with 16-bit hex value xxxx
\Uxxxxxxxx	Character with 32-bit hex value xxxxxxxx

```
rb"byte string"
```

Binary Ops

expr binop expr

Unary Ops

unop expr

Comparisons

expr comparison expr

<
>
==
>=
<=
<>
!=
in
not in
is
is not

```
class Circle:
  radius = 10

  def __eq__(self,other):
    return self.radius == other.radius

c1 = Circle()
c2 = Circle()

c1.radius = 3
c2.radius = 3

print (c1 is c2)  # prints False
print (c1 == c2)  # prints True
```

```
 3 <  4  < 5
(3 <  4) < 5
3 < (4  < 5)
```

```
1 == 1
```

```
1 == 1  # is True
```

```
1 == 1 is True
```

```
1 == 1 is True # is False
```

Bits

|

^

&

<<

>>

~

Arithmetic

+

-

*

/

%

//

**

Booleans & Conditionals

ontrue if *cond* else *onfalse*

expr and *expr*

expr or *expr*

```
not expr
```

Collections

Tuples

(1,2,3)

1,2,3

```
t = (1,2,3)
print (t[1]) # prints 2
```

```
t = (0,1,2,3,4,5,6,7,8,9)
print (t[2:5])
# prints (2,3,4)
```

Lists

```
[1,2,3]
```

```
t = [1,2,3]
print (t[1]) # prints 2
```

```
t = [0,1,2,3,4,5,6,7,8,9]
print (t[2:5])
# prints [2,3,4]
```

```
t = [0,1,2,3,4,5,6,7,8,9]
t[2:5] = [20,30,40,45]
print(t)
# prints [0,1,20,30,40,45,5,6,7,8,9]
```

Slices

```
t = [0,1,2,3,4,5,6,7,8,9]
print(t[0:9:2])
```

```
t = [0,1,2,3,4,5,6,7,8,9]
print(t[0:9:2])
# prints [0, 2, 4, 6, 8]
```

```
t = [0,1,2,3,4,5,6,7,8,9]
print(t[9:0])
```

```
t = [0,1,2,3,4,5,6,7,8,9]
print(t[9:0])
# prints []
```

```
t = [0,1,2,3,4,5,6,7,8,9]
print(t[9:0:-1])
```

```
t = [0,1,2,3,4,5,6,7,8,9]
print(t[9:0:-1])
# prints [9, 8, 7, 6, 5, 4, 3, 2, 1]
```

```
t = [0,1,2,3,4,5,6,7,8,9]
print(t[9:-1:-1])
```

```
t = [0,1,2,3,4,5,6,7,8,9]
print(t[9:-1:-1])
# prints []
```

```
t = [0,1,2,3,4,5,6,7,8,9]
print(t[9::-1])
```

```
t = [0,1,2,3,4,5,6,7,8,9]
print(t[9::-1])
# prints [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

```
t = [0,1,2,3,4,5,6,7,8,9]
print(t[::-1])
```

```
t = [0,1,2,3,4,5,6,7,8,9]
print(t[::-1])
# prints [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

```
t = [0,1,2,3,4,5,6,7,8,9]
print(t[::])
```

```
t = [0,1,2,3,4,5,6,7,8,9]
print(t[::])
# prints [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
t = [0,1,2,3,4,5,6,7,8,9]
print(t[-1])
```

```
t = [0,1,2,3,4,5,6,7,8,9]
print(t[-1])
# prints 9
```

```
t = [0,1,2,3,4,5,6,7,8,9]
print(t[-10])
```

```
t = [0,1,2,3,4,5,6,7,8,9]
print(t[-10])
# prints 0
```

```
t = [0,1,2,3,4,5,6,7,8,9]
print(t[-10:0])
```

```
t = [0,1,2,3,4,5,6,7,8,9]
print(t[-10:0])
# prints []
```

```
t = [0,1,2,3,4,5,6,7,8,9]
print(t[-10:None])
```

```
t = [0,1,2,3,4,5,6,7,8,9]
print(t[-10:None])
# prints [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
t = [0,1,2,3,4,5,6,7,8,9]
print(t[-1:-11:-1])
```

```
t = [0,1,2,3,4,5,6,7,8,9]
print(t[-1:-11:-1])
# prints [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

```
t = [0,1,2,3,4,5,6,7,8,9]
print(t[-1:3:-1])
```

```
t = [0,1,2,3,4,5,6,7,8,9]
print(t[-1:3:-1])
# prints [9, 8, 7, 6, 5, 4]
```

Slices work on strings too!

Dictionaries

$$\{\ key_1\texttt{:}\ value_1\texttt{, ..., }key_n\texttt{:}\ value_n\ \}$$

```
a = {'foo': 3, 'bar': 4}
print (a['foo']) # prints 3
```

Sets

$$\{\ value_1,\ \ldots,\ value_n\ \}$$

{1,2,3}

Lambda

```
lambda parameters: expression
```

```
f = lambda x: x
```

```
f(20)
```

```
f = lambda x: x

f(20) # == 20
```

```
f = lambda x,y: x + y
```

```
f(20,30)
```

```
f = lambda x,y: x + y
```

```
f(20,30) # == 50
```

```
   (((lambda f: (((f)((lambda f: ((lambda z: (((f)(((f)(((f)(((f)(((f)
    (z))))))))))))))))))((((((lambda y: ((lambda F: (((F)((lambda x:
((((((((y)(y)))(F)))(x)))))))))))((lambda y: ((lambda F: (((F)((lambda x:
 ((((((((y)(y)))(F)))(x)))))))))))((lambda f: ((lambda n: ((((((((((((
lambda n: (((((n)((lambda _: ((lambda t: ((lambda f: (((f)((lambda void:
(void))))))))))))))((lambda t: ((lambda f: (((t)((lambda void: (void)))))
   )))))))((((((lambda n: ((lambda m: (((((m)((lambda n: ((lambda f:
 ((lambda z: ((((((n) ((lambda g: ((lambda h: (((h)(((g)(f)))))))))))
((lambda u: (z)))))((lambda u: (u))))))))))))))(n))))))) (n)))((lambda f:
 ((lambda z: (z)))))))))((lambda _: ((((lambda n: (((((n) ((lambda _: ((
lambda t: ((lambda f: (((f)((lambda void: (void))))))))))))) ((lambda t:
   ((lambda f: (((t)((lambda void: (void))))))))))) (((((((lambda n:
((lambda m: (((((m)((lambda n: ((lambda f: ((lambda z: ((((((n) ((lambda
   g: ((lambda h: (((h)(((g)(f)))))))))))((lambda u: (z)))))((lambda u:
   (u))))))))))))(n))))))((lambda f: ((lambda z: (z))))))(n)))))))))
 ((lambda _: ((lambda t: ((lambda f: (((f)((lambda void: (void))))))))))
 ))((lambda _: ((lambda f: ((lambda z: (((f)(z)))))))))((lambda _: (((
 (((lambda n: ((lambda m: ((lambda f: ((lambda z: (((((m)(((n)(f)))))(z)
   ))))))))))(n)))(((f) (((((((lambda n: ((lambda m: (((((m)((lambda n:
 ((lambda f: ((lambda z: ((((((n) ((lambda g: ((lambda h: (((h)(((g)(f)
 ))))))))))((lambda u: (z)))))((lambda u: (u))))))))))))(n))))))(n)))
((lambda f: ((lambda z: (((f) (z)))))))))))))))))))))))(lambda x:x+1)(0)
```

Statements

Assignment

```
x = 10
print(x) # prints 10
```

```
(x,y) = [1,2]
print(x)
print(y)
```

```
(x,y) = [1,2]
print(x) # prints 1
print(y) # prints 2
```

```
(x,*y) = [1,2,3]
print (x)
print (y)
```

```
(x,*y) = [1,2,3]
print (x) # prints 1
print (y) # prints [2,3]
```

```
(x,*y,z) = [1,2,3,4,5,6]
print (x)
print (y)
print (z)
```

```
(x,*y,z) = [1,2,3,4,5,6]
print (x) # prints 1
print (y) # prints [2,3,4,5]
print (z) # prints 6
```

```
(x,*y,z,*w) = [1,2,3,4,5,6]
print (x)
print (y)
print (z)
print (w)
```

```
(x,*y,z,*w) = [1,2,3,4,5,6] # error!
print (x)
print (y)
print (z)
print (w)
```

Conditionals

```
if cond:
    statements
```

```
if cond:
    statements
elif cond:
    statements
```

```
if cond:
    statements
elif cond:
    statements
else:
    statements
```

```
while cond:
    statements
```

```
while cond:
    statements
else:
    statements
```

```
x = 4
while x > 0:
  print(x)
  x = x - 1
else:
  print("done")
```

```
x = 4
while x > 0:
  print(x)
  x = x - 1
else:
  print("done")
```

```
4
3
2
1
done
```

```
x = 4
while x > 0:
  print(x)
  x = x - 1
else:
  print("done")
```

?

```
x = 4
while x > 0:
  print(x)
  x = x - 1

print("done")
```

```
x = 4
while x > 0:
  print(x)
  x = x - 1
  if x == 0:
    break
else:
  print("done")
```

```
x = 4
while x > 0:
  print(x)
  x = x - 1
  if x == 0:
    break
else:
  print("done")
```

```
4
3
2
1
```

```
for params in expr:
    statements
```

```
a = [10,20,30,40]
for x in a:
    print(x)
```

```
a = [10,20,30,40]
for x in a:
  print(x)
# prints 10, 20, 30, 40
```

```
a = [10,20,30,40]
for x in a:
  print(x)
else:
  print("done")
```

```
a = [10,20,30,40]
for x in a:
  print(x)
else:
  print("done")
# prints 10, 20, 30, 40, done
```

Functions

```
def name(params):
    statements
```

```
def f(x,y,z):
  return x + y + z

f(1,2,3)
```

```
def f(x,y,z):
  return x + y + z

f(1,2,3) # == 6
```

Keyword args

```
def printme(arg1,arg2):
  print(arg1)
  print(arg2)

printme(arg2="me second",arg1="me first")
```

```
def printme(arg1,arg2):
  print(arg1)
  print(arg2)

printme(arg2="me second",arg1="me first")
# prints me first, me second
```

Optional args

```
def optarg(x,y=10):
  return x + y

print(optarg(10,20))
print(optarg(10))
```

```
def optarg(x,y=10):
  return x + y

print(optarg(10,20))  # prints 30
print(optarg(10))     # prints 20
```

Variable arity args

```
def vararity(x,*y):
  print(y)

vararity(1,2)
vararity(1,2,3)
vararity(1,2,3,4)
```

```
def vararity(x,*y):
  print(y)

vararity(1,2)     # prints (2,)
vararity(1,2,3)   # prints (2,3)
vararity(1,2,3,4) # prints (2,3,4)
```

Dictionary args

```
def dictargs(x,**y):
  print(y)

dictargs(4,ent=1701,mol=42)
```

```
def dictargs(x,**y):
  print(y)

dictargs(4,ent=1701,mol=42)
# prints {'mol': 42, 'ent': 1701}
```

Mixing args

```
def cluster(a,b=10,*c,**d):
  print(a)
  print(b)
  print(c)
  print(d)

cluster(13,"foo","bar","baz",opt1=10,opt2=30)
```

```
def cluster(a,b=10,*c,**d):
  print(a) # prints 13
  print(b) # prints foo
  print(c) # prints ('bar', 'baz')
  print(d) # prints {'opt1': 10, 'opt2': 30}

cluster(13,"foo","bar","baz",opt1=10,opt2=30)
```

Calling functions

```
args = [1,2,3]
def print3(a,b,c):
  print(a)
  print(b)
  print(c)

print3(*args)
```

```
args = [1,2,3]
def print3(a,b,c):
  print(a) # prints 1
  print(b) # prints 2
  print(c) # prints 3

print3(*args)
```

```
args = {'a': 10, 'b': 20, 'c': 30}
def print3(a,b,c):
  print(a)
  print(b)
  print(c)

print3(**args)
```

```
args = {'a': 10, 'b': 20, 'c': 30}
def print3(a,b,c):
  print(a) # prints 10
  print(b) # prints 20
  print(c) # prints 30

print3(**args)
```

Scope

```
g = 3.14

def f():
  print(g)

f()

print(g)
```

```
g = 3.14

def f():
  print(g) # prints 3.14

f()

print(g)   # prints 3.14
```

```
g = 3.14

def f():
  g = 6.28
  print(g)

f()

print(g)
```

```
g = 3.14

def f():
  g = 6.28
  print(g) # prints 6.28

f()

print(g)   # prints 3.14
```

```
g = 3.14

def f():
   g = 2*g
   print(g)

f()

print(g)
```

```
g = 3.14

def f():
  g = 2*g  # error: g used before definition
  print(g)

f()

print(g)
```

```
g = 3.14

def f():
  f = 2*g  # error: g used before definition
  g = 5
  print(g)

f()

print(g)
```

```
g = 3.14

def f():
   global g
   g = 2*g
   print(g)

f()

print(g)
```

```
g = 3.14

def f():
  global g
  g = 2*g
  print(g) # prints 6.28

f()

print(g) # prints 6.28
```

```
g = 3.14

def f():
  nonlocal g
  g = 2*g
  print(g)

f()

print(g)
```

```
g = 3.14

def f():
  nonlocal g # error: no nonlocal `g`
  g = 2*g
  print(g)

f()

print(g)
```

```
def outside():

  g = 3.14

  def f():
    nonlocal g
    g = 2*g
    print(g)

  f()

  print(g)

outside()
```

```
def outside():

  g = 3.14

  def f():
    nonlocal g
    g = 2*g
    print(g) # prints 6.28

  f()

  print(g) # prints 6.28

outside()
```

Lexical Scope

```
y = "global"

def make_procedure():
    y = "local"
    return lambda: y

f = make_procedure()

print(f())
```

```
y = "global"

def make_procedure():
    y = "local"
    return lambda: y

f = make_procedure()

print(f()) # prints local
```

```
y = "global"

def make_procedure():
  y = "local"
  def f(): return y
  return f

f = make_procedure()

print(f()) # prints local
```

Classes

Inheritance

```
class Animal:
  def speak(self): print(self.speech)

class Dog(Animal):
  speech = "woof"

d = Dog() ;
d.speak() ;   # prints woof
```

```
class Animal:
  def speak(self): print(self.speech)

class Dog(Animal):
  speech = "woof"

d = Dog() ;
d.speak() ;
```

Multiple inheritance

```
class Animal:
  def speak(self):
    print ("noise")

class Cat(Animal):
  def meow(self):
    print ("meow")

class Dog(Animal):
  def woof(self):
    print ("woof")
```

```
class CatDog(Cat,Dog):
  def speak(self):
    self.meow()
    self.woof()
    super().speak()

cd = CatDog()

cd.speak()
# prints
# meow, woof, noise
```

First-class classes

```
class Vehicle:
  fuel = 42

someclass = Vehicle

my_car = someclass()
print(my_car.fuel)

print(type(my_car))
```

```
class Vehicle:
  fuel = 42

someclass = Vehicle

my_car = someclass()
```

```
print(my_car.fuel)
# prints 42

print(type(my_car))
# prints <class '__main__.Vehicle'>
```

```
class Vehicle:
  fuel = 42

someclass = Vehicle

class Car(someclass):
  def refuel(self):
     self.fuel += 100

c = Car()
c.refuel()
print(c.fuel)
```

```
class Vehicle:
  fuel = 42

someclass = Vehicle

class Car(someclass):
  def refuel(self):
     self.fuel += 100

c = Car()
c.refuel()
print(c.fuel) # prints 142
```

```
class Ship:
  fuel = 1701

s1 = Ship()

s2 = type(s1)()

print (type(s2))


print (s2.fuel)
```

```
class Ship:
  fuel = 1701

s1 = Ship()

s2 = type(s1)()

print (type(s2))
# prints <class '__main__.Ship'>

print (s2.fuel)
```

```
class Ship:
  fuel = 1701

s1 = Ship()

s2 = type(s1)()

print (type(s2))
# prints <class '__main__.Ship'>

print (s2.fuel) # prints 1701
```

Classes: Scope & Identity

```
def make_class (x):

  class A:
    def get_x(self):
      return x

  return A

Alpha = make_class(10)
Beta = make_class(20)

print (Alpha)
print (Beta )

print (Alpha is Beta)

print (Alpha().get_x())
print (Beta() .get_x())
```

```
def make_class (x):

  class A:
    def get_x(self):
      return x

  return A

Alpha = make_class(10)
Beta = make_class(20)

print (Alpha) # prints <class '__main__.make_class.<locals>.A'>
print (Beta ) # prints <class '__main__.make_class.<locals>.A'>

print (Alpha is Beta)

print (Alpha().get_x())
print (Beta() .get_x())
```

```
def make_class (x):

  class A:
    def get_x(self):
      return x

  return A

Alpha = make_class(10)
Beta = make_class(20)

print (Alpha) # prints <class '__main__.make_class.<locals>.A'>
print (Beta ) # prints <class '__main__.make_class.<locals>.A'>

print (Alpha is Beta) # prints False

print (Alpha().get_x())
print (Beta() .get_x())
```

```
def make_class (x):

  class A:
    def get_x(self):
      return x

  return A

Alpha = make_class(10)
Beta = make_class(20)

print (Alpha) # prints <class '__main__.make_class.<locals>.A'>
print (Beta ) # prints <class '__main__.make_class.<locals>.A'>

print (Alpha is Beta) # prints False

print (Alpha().get_x())  # prints 10
print (Beta() .get_x())  # prints 20
```

Exceptions

```
try:
   statements
except exception-class:
   statements
```

raise *exception*

```
try:
   print (42)
except:
   print ("error!")
```

```
try:
   print (42)
except:
   print ("error!")
# prints 42
```

```
try:
   raise Exception()
except:
   print ("error!")
```

```
try:
   raise Exception()
except:
   print ("error!")
# prints error!
```

```
class A_Exception(Exception): pass
class B_Exception(A_Exception): pass

try:
  raise A_Exception()
except B_Exception:
  print ("got B")
except A_Exception:
  print ("got A")
```

```
class A_Exception(Exception): pass
class B_Exception(A_Exception): pass

try:
  raise A_Exception()
except B_Exception:
  print ("got B")
except A_Exception:
  print ("got A")
# prints "got A"
```

```
class A_Exception(Exception):
  def __init__(self,msg):
    self.msg = msg

try:
  raise A_Exception("a message")
except A_Exception as a:
  print ("got A: " + a.msg)
```

```
try:
  print("make a mess")
finally:
  print("clean up")
```

```
def f():
    try:
        return 20
    finally:
        print("do this")

x = f()
print(x)
```

```
def f():
  try:
    return 20
  finally:
    print("do this")

x = f()  # prints do this
print(x) # prints 20
```

```
def f():
    try:
        return 20
    finally:
        return 30

x = f()
print(x)
```

```
def f():
  try:
    return 20
  finally:
    return 30

x = f()
print(x) # prints 30
```

Comprehensions

```
[i for i in range(1,4)]
```

```
[1, 2, 3]
```

```
[(x,y) for x in [1,2,3] for y in [4,5,6]]
```

```
[(1, 4), (1, 5), (1, 6),
(2, 4), (2, 5), (2, 6),
(3, 4), (3, 5), (3, 6)]
```

```
{ i for i in range(1,4) }
```

```
{1, 2, 3}
```

```
{ k: k + 1 for k in range(0,4) }
```

```
{0: 1, 1: 2, 2: 3, 3: 4}
```

Generators

```
((x,y) for x in [1,2,3] for y in [4,5,6])
```

```
<generator object <genexpr> at 0x102d5cdc0>
```

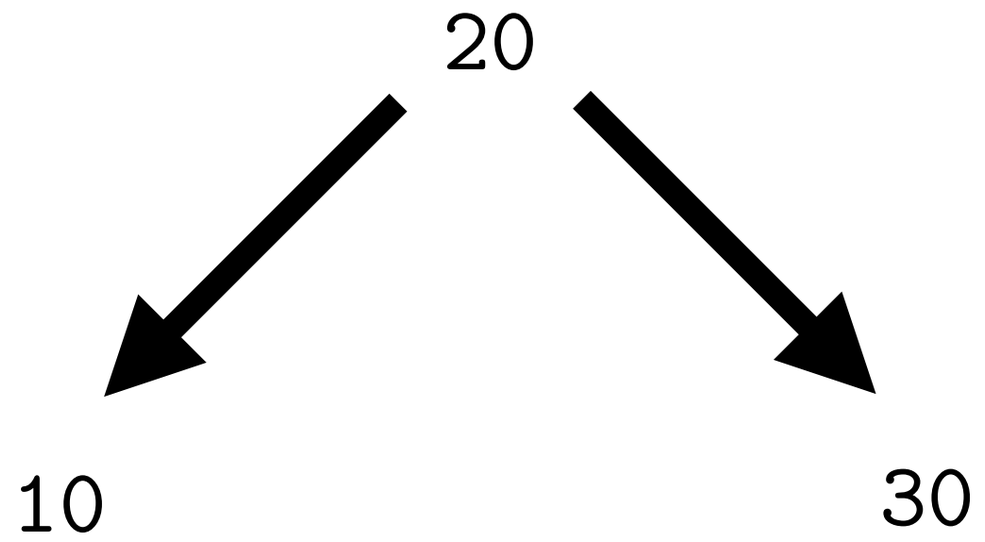
```
g = ((x,y) for x in [1,2,3] for y in [4,5,6])
for p in g:
  print (p)
```

(1, 4)
(1, 5)
(1, 6)
(2, 4)
(2, 5)
(2, 6)
(3, 4)
(3, 5)
(3, 6)

yield

```
class BSTNode:
  left = None
  right = None
  value = None

  def __init__(self,left,value,right):
    self.left = left
    self.right = right
    self.value = value
```

```
def inorder_bst(bst):
  if bst is ():
    return

  for v in inorder_bst(bst.left):
    yield v

  yield bst.value

  for v in inorder_bst(bst.right):
    yield v
```


20
10
30

```
tree = BSTNode(BSTNode((),10,()),
               20,
               BSTNode((),30,()))
```

```
for v in inorder_bst(tree):
    print(v)
```

```
for v in inorder_bst(tree):
    print(v)
# prints 10 then 20 then 30
```

```
def inorder_bst(bst):
  if bst is ():
    return

  for v in inorder_bst(bst.left):
    yield v

  yield bst.value

  for v in inorder_bst(bst.right):
    yield v
```

```
def inorder_bst(bst):
    if bst is ():
        return
    yield from inorder_bst(bst.left)
    yield bst.value
    yield from inorder_bst(bst.right)
```

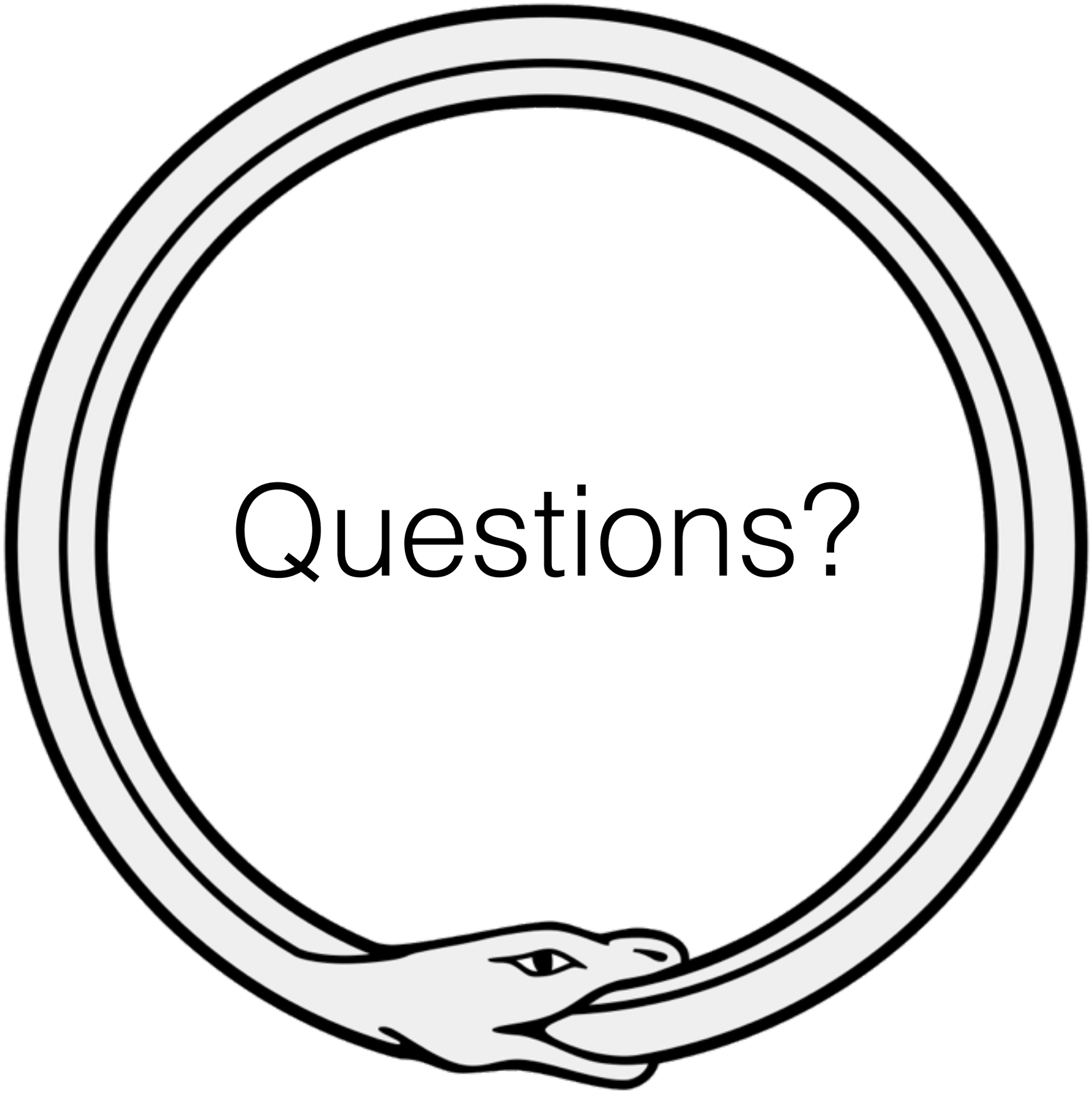
Decorators

```
def fib(n):
    if n == 0:
        return 1
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

print (fib(32))
```

```
@dec def name(params): suite
```

```
def memoize(f):
  table = {}
  def memo_f(x):
    if x in table:
      return table[x]
    else:
      y = f(x)
      table[x] = y
      return y
  return memo_f
```

```
@memoize
def fib(n):
  if n == 0:
    return 1
  elif n == 1:
    return 1
  else:
    return fib(n-1) + fib(n-2)

print (fib(32))
```


Questions?