From turtle import \*

Def init():

Setup(600,600)

Setworldcoordinates(-1,-1,21,21)

Lt(90)

Def drawSpiral0():

Pass

Def drawSpiral1():

Forward(1)

Rt(90)

Def drawSpiral2():

Fd(2)

Rt(90)

Fd(1)

Def drawSpiral(segments):

If(segments==0):

Pass

Else:

Fd(segments)

Rt(90)

drawSpiral(segments-1)

Init()

2drawSpiral1()

Tail recursion:

Exe diag:

drawSpiral(seg==3)

#segs<>0

Fd(3)

Rt(90)

drawSpiral(seg==2)

seg<>0

fd(2)

rt(90)

drawSpiral(seg==1)

#seg<>0

Fd(1)

Rt(90)

drawSpiral(seg==0)

pass

\*\*\*\*\*\*

Pseudocode parts

Define drawspiral(segs)

Repeat until done

If seg==0:

Done

Else:

Fd(seg)

Rt(90)

Replace segs with segs-1

Seg=seg-1

Variable table:

\*Keep track of variables, where in memory the values assoc. with variables are being stored

\*references to locaitons in computer memory

X=z+x >>>evaluate expression on right side, assign reslt to variable on left hand side

X refers to address holding value that right side evaluated to

Add to spirals

Def draw Iter(seg):

whileTrue:

if (seg)==0:

break—used to break you out of loops

else:

fd(seg)

rt(90)

seg=seg-1

def drawIter2(seg):

while(seg>0):

fd(seg)

rt(90)

seg=seg-1

drawIter()

drawIter2()

docstrings sometimes have f:d>r

nonetype returns nothing

function with no end statement

suncfion say return)

just return nonetype

tail recursion for fruitful functions

fed fact(n):

if (n==0):

return1

else:

return n \* fact(n-1)

def fib(n)

if n==0:

return0

elfin==1

return1

else:

return fib(n-1)+fib(n-2)

convert fact to tailrecursive:::::

def factAccum(n,a) #a stores accumulated result, allows computing of statements and have return of answer as last thing done.

if n==0:

return a

else:

return factAccum(n-1,n\*a)

subst trace

factAccum(4,1)

\*always start with a of 1

=factAxxum(3,4)=factAccum(2,12)=factAccum(1,24)=factAccum(0,24)=24

Map to an iterative version fact(n)=factAccum(n,1)🡪code

defFactAccumIter(n,a)

whileTrue:

if n==0:

returna

else:

a=n\*a

n=n-1

def factIT(n):

a=1 #init. Accumulator

while (n>0):

a=n\*a

n=n-1

return a

def(fib(n):

if(n==0:

return0

elif n==1:

return 1

else:

return fib(n-1) + fib(n-2)

def fibAccum(n,a,b):

ifn n==0:

return a

elif n==1:

return b

else:

return fibAccum(n-1,b,a+b)

fib(n)=fibAccum(n,0,1)

def fibIt1(n,a,b):

whileTrue:

if n ==0:

return a

if n ==1:

return b

else:

n=n-1

new a = b

new b == a+b

a=newa

b=newb

a 🡪 new a is b

b 🡪 new b is a+b

tempx=x

x=y

n=tempx

def fibIt(n) #example should work

a=0

b=1

while(n>1:

n=n-1

newa=b

newb=a+b

a=newa

b=newb

if n==0:

return a

else:

return b

fib(4)

fib(accum(4,0,1)

=fibAccum(3,1,1)

=fibAccum(2,1,2)

=fibAccum(1,2,3)

=3

\*\*\* \*\*\* \*\*

Need to know

What it means to be tail recursive: no work done after recursive call

The point of tail recursion is putting a function in it, maps to interative very easily

Comfortable writing either recursive or iterative functions

Don’t worry about

Don’t worry about converting from nontail to tail recursive

Don’t worry about converting from tail recursive to iterative

\*\*\*

X=10

Global variable typically defined at the beginning of the file and defined outside of any function. Harder to keep track of whats going on in the code. Any function can access this variable. We try to avoid this, make it difficult to understand how the code works because we have to track down where the variable is and what the value is supposed to be. Changing it globally later on also makes it more difficult to piece the code together to see how it works and the information might need to be modified.

Def fool():

Y=x+3

Print(“value of y is” + str(y))

Def foo2():

X=50

Print(“Inside foo2, x is” + str(x))

Foo1()

Foo2()

Print(“after oo2” + str(x))

Def foo3():

Z=x+5

X=20

#cant define variable after equation in local, then call it and use global and local and error

Foo3()

Def foo4()

Global x

Z=x+5

X=20

Print(“x is now” + str(x))

Foo4()

Print(“after foo 4, x is” + str(x))

Deffoo5():

Localvar=625

Foo6(localvar)

Print(“after colling foo6, localvar has value” + str(localvar))

Def foo6(localvar):

Localvar=987

Print(“inside foo6, localvar has value of “ + str((localvar))

Foo5()

Foo6()

Start at t=0, the nincrease by .01, this draws the larger picture