COSC2429 Introduction to Programming

Functions & Iterations

Outline

- Function calls
- Program vs. function
- Divide-and-conquer strategy
- Function definitions
- Fruitful functions
- Global vs. local variables
- Calling functions
- Example: A turtle bar chart
- **for** loop revisited
- **while** loop
- Using **for** loop or **while** loop?
- Example: Newton’s method to compute square root

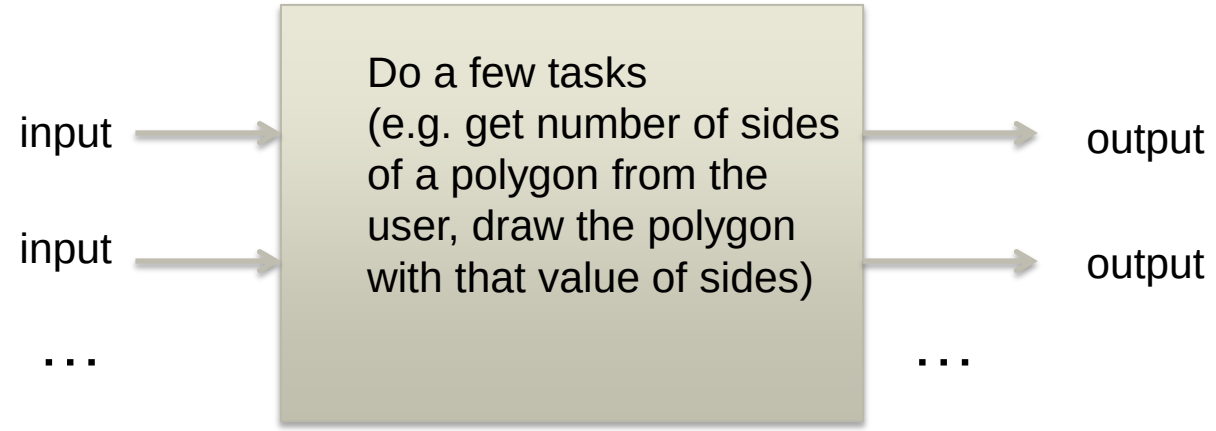
Function calls

- So far we have used functions (or methods) written by someone else by calling those functions.
- Can you identify the function calls in the following program?

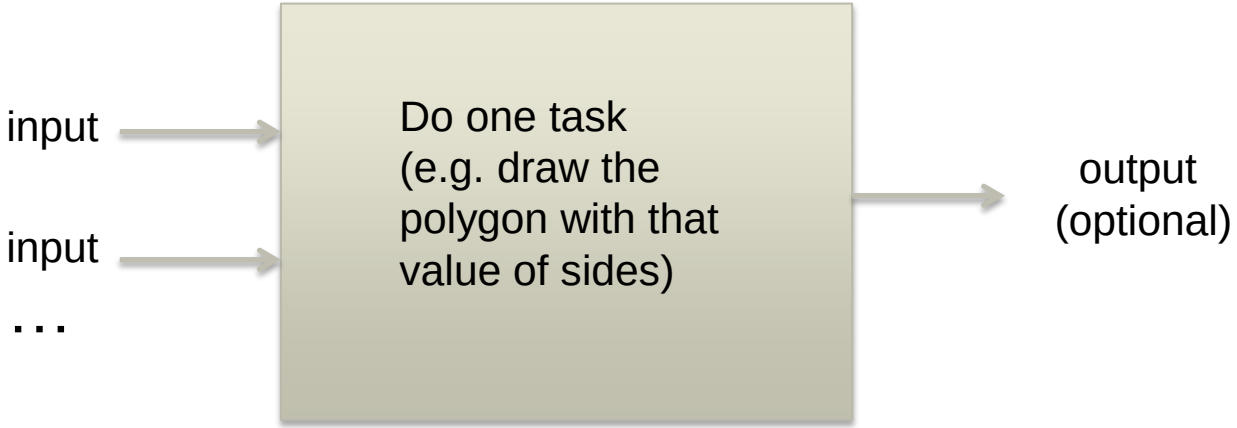
```
import turtle

# Setup a drawing window and a turtle
win = turtle.Screen()
win.bgcolor("lightgreen")
tess = turtle.Turtle()
tess.color("blue")
tess.pensize(3)

# Draw something
tess.forward(50)
tess.left(120)
tess.forward(50)

win.exitonclick()
```

- We can also **create our own functions** to be used later.

Program vs. function



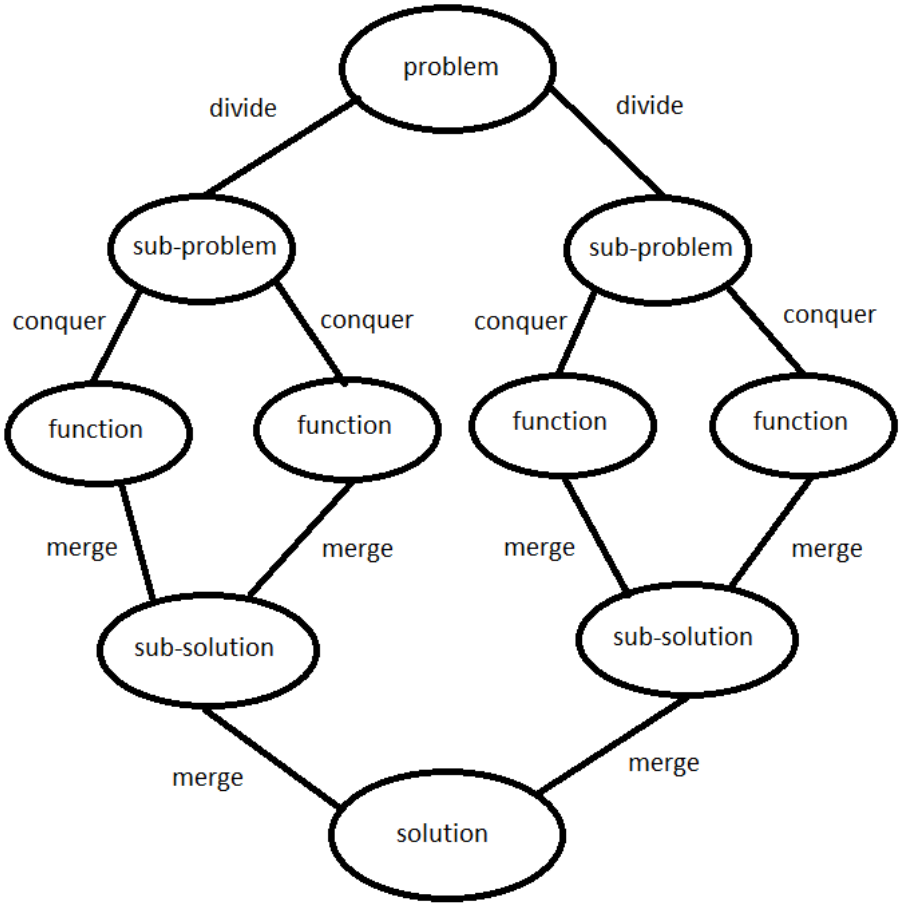
Program



Function
(aka mini-program)

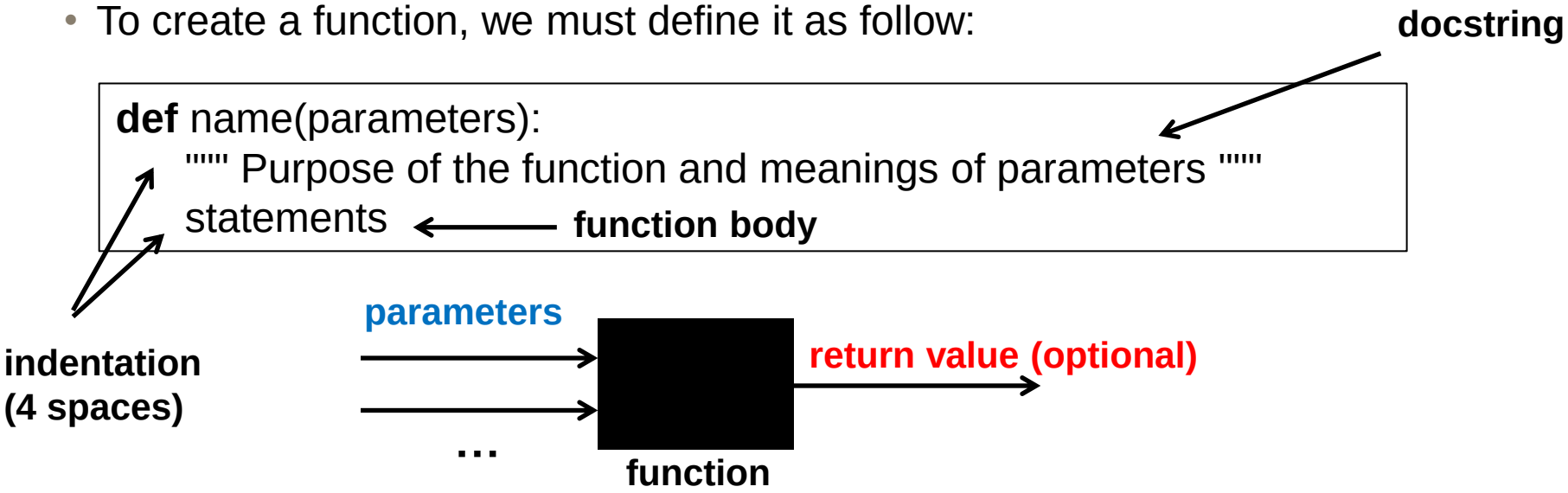
Divide-and-conquer strategy

- A problem can be divided into a number of sub-problems
- Each sub-problem can be solved by creating one or more functions
- Each function should do **only one task** and **do it really well** => **specialization**
- The solution of the original problem can be created by **calling the functions** of the sub-problems and **combining their results**
- After a function is created, you can call it as many times as you want => **reuse**



Defining a function

- In Python, a **function** is a named sequence of statements that belong together. Each function should do **only one thing** and **do it really well**.
- To create a function, we must define it as follow:



- The parameters are known as **formal parameters** which describe the inputs of the function.
- When you call a function, you must provide the actual values for these formal parameters.
- These values are often regarded as **arguments** or **actual parameters**

Defining a function

- Suppose we're working with turtles and a common operation we need is to draw squares. It makes sense if we don't have to duplicate all the steps each time we want to make a square.
- Thus it makes sense to define a function to draw a square. We will need to provide two pieces of information for the function to do its work: a turtle to do the drawing and a size for the side of the square.

```
import turtle

# Function definition appears before the function is called
def draw_square(t, sz):
    """ Turtle t draws a square of side sz """
    for i in range(4):
        t.forward(sz)
        t.left(90)

# Main program starts after all function definitions
win = turtle.Screen()
win.bgcolor("lightgreen")
tito = turtle.Turtle()

# Call the function to draw the square with actual values for turtle and side
draw_square(tito, 50)
win.exitonclick()
```

t and sz are formal parameters

docstring describes the purpose of the function

tito and 50 are actual parameters (or arguments)

Defining a function

- Once the function is defined, we can call it **as many times as we want** with whatever actual parameters we like.
- Let's change the **draw_square** function a little and we get tito to draw 15 squares with some variations.

```
import turtle

# Function definition
def draw_square(t, sz):
    """“ Turtle t draws a multi-colour square of side sz."""
    for a_color in ['red', 'purple', 'hotpink', 'blue']:
        t.color(a_color)
        t.forward(sz)
        t.left(90)

# Main program starts from here
# Set up a drawing window and a turtle
win = turtle.Screen()
win.bgcolor("lightgreen")
tito = turtle.Turtle()
tito.pensize(3)

# Draw a mystery picture made up by 15 squares
size = 20                    # size of the smallest square
for i in range(15):
    draw_square(tito, size)
    size = size + 10         # increase the size for next time
    tito.forward(10)         # move alex along a little
    tito.right(18)           # and give him some extra turn

win.exitonclick()
```

Fruitful functions

- A function that returns a value is called a **fruitful function.**
- The function draw_square that we had earlier doesn't return a value thus it is called a **non-fruitful function**.
- How do we write our own fruitful function? Here is an example.

```
# Function definition
def square(x):
    """ Calculate and return x² """
    y = x * x
    return y                        # return statement

# Main program
to_square = 10
result = square(to_square)   # need a variable to store the returned value
print("The result of ", to_square, " squared is ", result)
```

Global vs. local variables

- Variables and parameters defined in a function are **local variables**, i.e. they only exist inside the function and you can't use them outside the function.
- Consider the following example:

```
# Function definition
def square(x):
    """ Calculate and return x2 """
    y = x * x              # x and y are local variables
    return y

# Main program
z = square(10)         # z is a global variable that gets the value function square returns
print(z)
print(y)
```

- Each call of the function creates new local variables, and **their lifetimes expire when the function returns to the caller**.
- Variables and parameters defined in the main program are **global variables**.

Global vs. local variables

- It is legal for a function to access a global variable. However, this is considered a **bad practice** and should be avoided.
- Look at the following nonsensical variation of the square function:

```
# Function definition
def bad_square(x):
    y = x ** power              # bad form, DO NOT access global variable
    return y

# Main program
power = 2
result = bad_square(10)
print(result)
```

- The **correct way** to write this function would be to **pass power as an argument** to the function. That means you should add another parameter to the function in order to receive the value of the global variable power.
- For practice, please rewrite the above program.

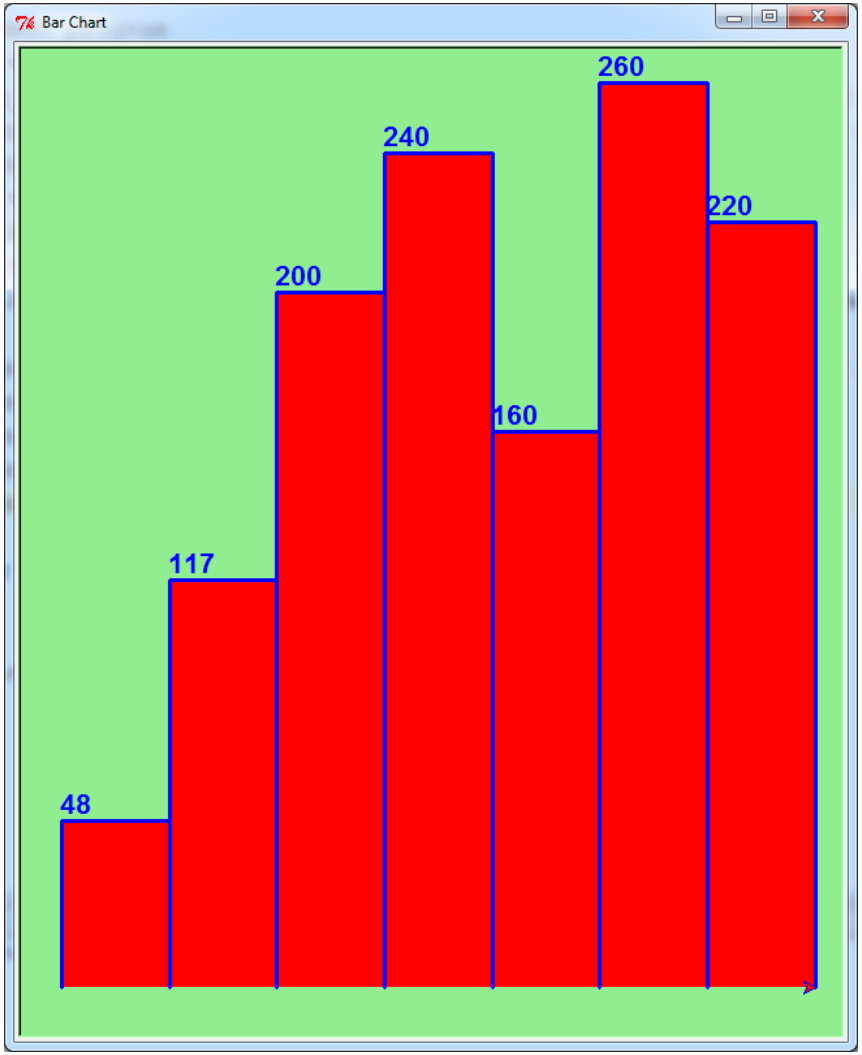
Calling functions

- Each of the functions we write can also be used and called from other functions we write => allow us to efficiently take a large problem and break it down into a group of smaller problems.
- This process of breaking a problem into smaller sub-problems is called **functional decomposition**.
- Here's a simple example of functional decomposition using two functions.

```
# Function definitions
def square(x):              # skip function's docstring for simplicity
    y = x * x
    return y

def sum_of_squares(x, y, z):        # skip function's docstring for simplicity
    a = square(x)
    b = square(y)
    c = square(z)
    return a + b + c

# Main program starts here
a = -5
b = 2
c = 10
result = sum_of_squares(a, b, c)
print(result)
```

Example: A turtle bar chart

- Write a program to draw a bar chart that looks exactly as the below picture where the numbers representing the heights of the bars in pixels.



```
import turtle

# Function definition
def draw_bar(t, ht):
    """ Get turtle t to draw a bar of height ht. """
    t.begin_fill()              # start filling this shape
    t.left(90)
    t.forward(ht)
    t.write(ht, font=("Arial", 16, "bold"))          # write the value of height in font Arial, 16 px and bold
    t.right(90)
    t.forward(40)
    t.right(90)
    t.forward(ht)
    t.left(90)
    t.end_fill()                # stop filling this shape

# Main program starts here
win = turtle.Screen()
win.bgcolor("lightgreen")
win.title("Bar Chart")
tess = turtle.Turtle()
tess.color("blue")
tess.fillcolor("red")
tess.pensize(3)

# Move tess to the left to make the bars centre
tess.up()
tess.goto(-100, 0)
tess.down()

# Use a loop to draw 7 bars
for height in [48, 117, 200, 240, 160, 260, 220]:
    draw_bar(tess, height)

win.exitonclick()
```

for loop revisited

- Recall that the for loop processes each item in a list. Each item in turn is assigned to the loop variable, and the body of the loop is executed.

```
for friend in ["Joe", "Amy", "Brad", "Angelina", "Zuki", "Thandi", "Paris"]:
    print(Hi", friend, "Please come to my party on Saturday")
```

- Here is another example that calculate the sum of integers:

```
# Function definition
def sum_to(n):
    """ Return the sum of 1 + 2 + 3 ... + n """
    sum = 0
    for num in range(1, n + 1):
        sum = sum + num
    return sum

# Main program
print("Sum of 1 to 4 is", sum_to(4))
print("Sum of 1 to 1000 is", sum_to(1000))
print("Sum of 1 to 1000000 is", sum_to(1000000))
```

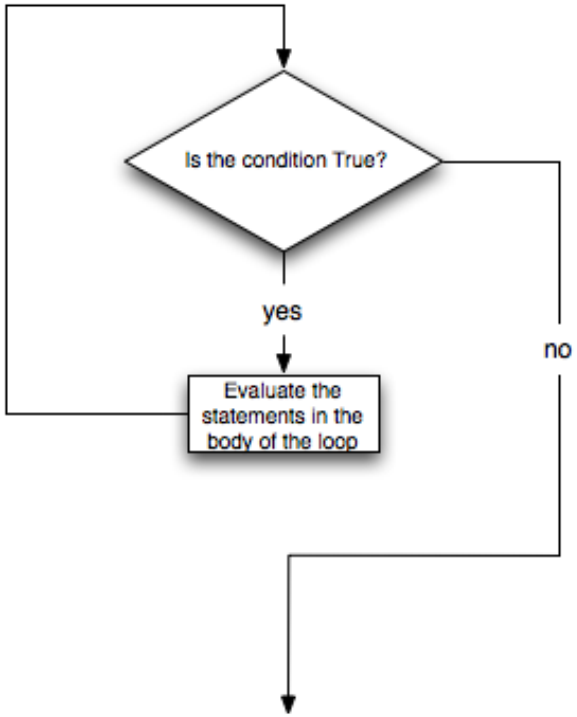
while loop

- **while** loop is another Python statement that can also be used to build an iteration. It provides a much more general mechanism for iterating.
- The syntax for a **while** loop looks like this:

```
while expression:          # expression is True for any non-zero value
    statements             # executed if condition evaluates to True
```

indentation (4 spaces)

- We can use the while loop to create any type of iteration we wish, including anything that we have previously done with a for loop.



while loop

- Here is another version of the sum_to program using **while** loop:

```
# Function definition
def sum_to(n):
    """ Return the sum of 1 + 2 + 3 ... + n """
    sum = 0
    num = 1
    while num <= n:
        sum = sum + num
        num = num + 1
    return sum

# Main program
print("Sum of 1 to 4 is", sum_to(4))
print("Sum of 1 to 1000 is", sum_to(1000))
print("Sum of 1 to 1000000 is", sum_to(1000000))
```

- What you notice here is that while loop requires a little bit more work than the for loop.

Using **for** loop or **while** loop?

- **while** loop is more generic than **for** loop
  - Any code using a **for** loop can be written using a **while** loop
- But sometimes it's more "natural" to use **for** loop. When?

Example: Newton's method to compute square root

- Loops are often used in programs that compute numerical results by starting with an approximate answer and iteratively improving it.
- For example, one way of computing square roots is Newton's method. Suppose that you want to know the square root of n. If you start with almost any approximation, you can compute a better approximation with the following formula:

```
better =  1 / 2 * (approx + n / approx)
```

- Here is an implementation of Newton's method that requires two parameters:

```
# Function definition
def newton_sqrt(n, how_many):
    approx = 0.5 * n
    for i in range(how_many):
        better_approx = 0.5 * (approx + n/approx)
        approx = better_approx
    return better_approx

# Main program to test the function
print(newton_sqrt(10,3))
print(newton_sqrt(10,5))
print(newton_sqrt(10,10))
```