Wednesday, Sept. 23 Notes   Nicole White

NOTE: **%** operator calculates the remainder of integer division
e.g. 10 % 3 = 1 (10/3 = 3 remainder 1)
This may be useful for Assignment 1

Last day: income tax calculations (tax_payable function)

```
bp1 = 10320     # Upper limit of lowest tax bracket
rate1 = 0        # Tax rate for lowest tax bracket
.            # This function is written out in full in the notes for Sept. 21
.
def tax_payable(income):
    if income <= bp1:
        return income * rate1
    elif income <= bp2:
        return b1 * rate1 + (income - bp1)*rate2
    .
    .
    else:
        return bp1*rate1 + (bp2 - bp1)*rate2 + (bp3 - bp2)*rate3 + \
        (bp4-bp3)*rate4 + (income - bp4)*rate5
```

This function becomes very complicated -> it would be very difficult to detect errors

- However, defining all of these calculations as constants clutters up the space and has little meaning to anyone other than the writer of the program
- SOLUTION: Declare **local constants** (only visible within the constraints of the program)

```
def tax_payable(income):
    max1 = bp1*rate1                        # Max tax paid in the first tax bracket
    max2 = max1 + (bp2 - bp1)*rate2     # Max paid in second tax bracket
    max3 = max2 + (bp3 - bp2)*rate3     # Max paid in third bracket
    max4 = max3 + (bp4 - bp3)*rate4     # Max paid in fourth bracket

    if income <= bp1:
        return income*rate1
    elif income <= bp2:
        return max1 + (income - bp1)*rate2
    elif income <= bp3:
        return max2 + (income - bp2)*rate3
    elif income <= bp4:
        return max3 + (income - bp3)*rate4
    else:
        return max4 + (income - bp4)*rate5
```

- Now, the formula doesn't get and longer as you move up in tax brackets (this is easier to read and takes up less space)
- We can have more confidence in the program, because as long as max# values are calculated correctly in their definitions, they will return correct answers throughout the entire program
- The constants max1/2/3/4 are not visible/have no meaning outside the definition of the function tax_payable

```
e.g. >>> max1
         error message ('name 'max1' is not defined')
```

A function declaration creates a new SCOPE

- the scope of a constant is the part of the program in which it is visible
- the constants we declared at the beginning (bp1, rate1, etc) are declared *outside* the function and have **global scope** (they can be seen everywhere in the program from their point of definition onward)
- constants defined *inside* the (function) definition have **local scope** (local to the function), and their scope begins at their point of definition and ends at the end of the function
    - Trying to access a local constant outside of its defined function will return an error

Local vs. Global constants:

```
>>> a = 5
>>> def f():    # a is defined both globally (5) and locally (7) --> there are 2 different a's
        a = 7
        return a

>>> a
5
>>> f()
7
```

This means that within the function f(), access to the global constant 'a' is *shadowed* by the local constant 'a' (the local constant **outscopes** the global constant)

- references to the constant 'a' inside the function f() resolve to the closest definition (the local one) and the global definition becomes invisible
- there is a way around this: within the function it is possible to ask for the global constant (next class)

EXERCISE: To graduate, you need a >= 50% average if you enrolled in 2007 or later, and at least 60% if you enrolled before 2007. Write a function that takes an enrollment year and average and determines whether a student graduates

```
def graduates(year, avg):
    if year >= 2007:
        if avg >= 50:
            return True
        else:
            return False
    else:
        if avg >= 60:
            return True
        else:
            return False
```

Another, more succinct way is to use Boolean operators:

```
def graduates(year, avg):
    if year >= 2007 and avg > 50:
        return True
    elif year < 2007 and avg >= 60:
        return True
    else:
        return False
```

AND: A Boolean operator which takes two statements and returns true if the values of both statements are true (this collapses the nested if statements we used above)

Another way:

```
def graduates(year, avg):
    if year >= 2007 and avg >= 50:
        return True
    elif avg >=60:      # if inputs fail the first branch, this can only return True if the
        return True     # year is LESS than 2007 (avoid redundancy)
    else:
        return False
```

Another way:

```
def graduates(year,avg):
    if (year >= 2007 and avg >= 50) or avg >= 60:
        return True
    else:
        return False
```

Another way:

```
def graduates(year,avg):
        return (year >= 2007 and avg >=50) or avg >= 60
```

- **General trick:** keep your code short!
    - if at any time you have:

```
        if condition:
                return True
        else:
                return False
```

you can *always* replace it simply with:

```
        return condition
```

(this returns True or False about the condition)

New Boolean operators:

AND: e.g. X and Y is true when X and Y are both true, and false otherwise

OR: e.g. X or Y is true when at least one of X or Y is true, and when both are true (returns False only if X and Y are both false)

Note: Boolean ‘or’ does not match common English usage (typically in English, ‘or’ means X or Y, but not both)

In Python, X or Y means X or Y or both