# Weeks 7 and 8

## 0.1 Outline

- Using an IDE for developing code in modules.
- Practice with recursion and designing recursive functions.
- Introduction to Code Testing (using `pytest`)

## 0.2 IDEs for Code Development

While Jupyter notebooks are a great way to experiment with code snippets and to provide annotations and explanatory commentary on the code, it is usually better to develop the code in the form of Python modules (`.py` files).

Once modules are defined, their namespaces (defined objects, functions, classes etc.) can be `import`ed in other modules or even in a Jupyter notebook!

### 0.2.1 Popular IDEs

There are several Integrated Development Environments (IDEs) where Python code for modules that are part of a project or package can be **developed**, **executed**, **debugged** and **tested**. IDEs also provide support for formatting the code, in a manner similar to what the notebook code cells do.

Among the most popular IDEs are the following:

- `IDLE`: the native, frills-free IDE that comes bundled with every Python installation
- `PyCharm`: a proprietary IDE
- `VS Code`: an industry standard IDE
- `Spyder`: part of the `Anaconda` distribution but can also be installed separately.

## 0.3 Recursion

A recursive function is defined in terms of **itself**; more specifically:

- it is defined *without further self-reference* for the **base case(s)** of the argument, viz. when the arguments are small enough.
- for larger arguments, it is defined in terms of its values on specific **smaller** arguments.

Note

Most recursive functions have base cases associated with the small integers 0, 1, 2 etc. The larger arguments are associated with larger integers: see, for example, the recursive function definition below.

### 0.3.1 Example 1: Factorial

Consider the **factorial** function defined for all $n \geq 0$ as:

$$f(n) = n \cdot (n-1) \cdot (n-2) \dots 1$$

Observe that the definition is understood by a *human* reader as saying: Form the product of successive descending numbers starting from $n$, then $(n-1)$, then $(n-2)$ **and so on** going down to 1!

A recursive specification would instead say that $f(1)$ equals 1 for the base case when $n = 1$. For larger values of $n$, we multiply $n$ with the result of the *recursive* computation $f(n-1)$ to obtain $f(n)$. Here is what the code might look like:

```python
def f(n: int) -> int:
    """Returns the factorial of n >= 1
    """
    if n == 1:
        return 1
    return n * f(n-1)
```

Two observations can be made about this example, one generic and one specific.

- **Specific:** The function exhibits *tail-recursion*, i.e. the stack of successive recursive function calls **unwinds** completely after the base case has been reached! Such function definitions can easily be converted to **loops**! See the code below:

```python
def f(n: int) -> int:  # Alternative factorial definition
    """Returns the factorial of n >= 1"""
    result = 1
    for num in reversed(range(1, n+1)):
        result = num * result
    return result
```

- **Generic:** Recursive calls are always made to **smaller** arguments. In this example, it is clear because we are dealing with numbers. In general, the idea of *smaller* extends to any collection of values that can be **well-ordered**, namely, values that are totally ordered and have a **least** element among any subset of elements. See the example below.

### 0.3.2 Example 2: Reversing a string

Let's define a recursive Python function to **reverse** a string. It is based on the observation that the reverse of a string named `text` is the concatenation of the last symbol `text` and the **reverse** of the string obtained by removing the last character of `text`. Note that this latter string is *smaller*, viz. it has length 1 less than the length of `text`. The **smallest** string is clearly well defined: it is the empty string and happens to be its own reverse.

```python
def reverse(text: str) -> str:
    """Returns the string obtained by reversing the symbols of text."""
    if len(text) == 0:
        return text
    return text[-1] + reverse(text[:-1])
```

### 0.3.3 Exercises

Define the following functions recursively. *Do not look at the code until you have tried it yourself*!

1. Count the number of transitions in a bitstring, i.e., the number of times we see an instance of `01` or `10` in a string containing just 0s and 1s.

```python
def n_transitions(bitstring: str) -> int:
    """Returns the number of transitions in a non-empty bitstring.

    A transition is an instance of `01` or `10`
    """
    if len(bitstring) == 0:
        return 0
    l_bit = bitstring[0]
    transition_bit = '0' if l_bit == '1' else '1'
    posn = bitstring.find(transition_bit)
    if posn == -1:  # no transition found in bitstring
        return 0
    # Count transitions from posn onwards recursively and add 1
    return 1 + n_transitions(bitstring[posn:])
```

3. Reverse the "words" in a string, e.g., `recursion is easy` becomes `easy is recursion`. You can assume that any maximal sequence of characters without a whitespace character is a "word".

### 0.3.4 Example 3: Recursion on one argument

Suppose we wish to define the multiplication of two integers **without** being able to use the multiply operator, but only using addition or subtraction.

Let's see the reasoning for non-negative integers, and we can then generalize to all integers. There are two arguments, call them `a` and `b`. We will recurse **only on** `b`!

- What is the simplest **base** case, i.e., where we **do not have to rely** on any mathematical operation for the result?
- What is the simplest *general* form of the recursive case that allows as to multiply `a` and `b` with just addition and the result of a recursive call with a smaller second argument (in place of `b`)?

Once you have thought through these questions, try to see if you write a recursive definition based on your answers. One possible solution is below:

```python
def multiply(a: int, b: int) -> float:
    """Returns a * b.
    """
    if b == 0:
        return 0
    return a + multiply(a, b-1)
```

Here is a similar example of recursing just on one argument. Suppose we wish to compute the result of raising a floating point number to an non-negative integer power.

```python
def exponentiation(base: float, exponent: int) -> float:
    """Returns base ** exponent.
    """
    if exponent == 0:
        return 1
    return base * exponentiation(base, exponent-1)
```

### 0.3.5 Example 4 (Harder): Partitions of a set into $k$ parts

A common combinatorial structure is the collection of **k-partitions of a set of** $n$ objects. In any such partition, every element of the set appears in exactly one part of the partition. For example, we can partition the letters "abcd" into 2 parts in the following three ways: "ab" and "cd"; "ac" and "bd"; "ad" and "bc". Note that there is no ordering of letters within a part and no ordering among the parts: the partition with parts "ab" and "cd" is the same as that with parts "dc" and "ba".

Let's assume that we are forming partitions of the symbols in a given string with unique symbols (these are the objects in the partitions). To make things even more concrete, each part will be a non-empty string, and each partition will be a tuple of such parts. Finally, any symbols that appear in a part will appear *in the same order* as in the given string.

Thus, we can say unambiguously that we want our function to create a set containing the seven partitions:

`('a', 'bcd'), ('ab', 'cd'), ('abc', 'd'), ('ac', 'bd'), ('acd', 'b'), ('ad', 'bc'), ('abd', 'c')`

when given the arguments `'abcd'` and 2, i.e. when it is asked to produce the set of all partitions of size 2 of the four symbols.

Note that there are **two arguments** here so we have to be careful about using an *ordering* of the combined arguments! We can think of the arguments as a tuple: the first a string of some finite length $n$ (the number of symbols/elements of the set) and the second, an integer $k$ indicating the number of parts in a partition.

The recursion here comes from a combinatorial argument: any partition of the set of objects $b_0, b_1, \ldots b_{n-1}$ into $k > 1$ parts must fall into one of two **disjoint** categories:

1. there is a part which **only contains** $b_0$: if we can obtain **recursively**, the set of partitions of $b_1, b_2, \ldots b_{n-1}$ into $(k-1)$ parts, we can augment each such partition by adding an additional part that only contains $b_0$.
2. every part that contains $b_0$ also contains another object: if we can obtain **recursively**, the set of partitions of $b_1, b_2, \ldots b_{n-1}$ into $k$ parts, we can augment each such partition in $k$ ways by inserting $b_0$ into one of the $k$ parts.

Look carefully at the parameters of the recursive calls, and you observe that in the first case, the call will have as first parameter a *smaller* set with $(n-1)$ elements and as second parameter, a *smaller* number of parts $(k-1)$. On the other hand, in the second case, the recursive call will have as first parameter a *smaller* set with $(n-1)$ elements and as second parameter, the *same* number of parts $k$.

If we think of the values $n$ and $k$ as forming a tuple, we see that in **both cases**, we are making a recursive call that features a **smaller** tuple, where the notion of smaller has to do with **lexicographic** ordering: e.g. (4,2) is larger than both (3,1) and (3,2); and in turn, (3,2) will be larger than (2,1) and (2,2) etc.

A little more thought should convince you that the recursion can **stop** when $k = 1$ or when $n = k$. In the first instance, we just obtain the one-part partition with all the elements, and in the second, we obtain an $n$-part partition with singleton elements as parts!

The code that formalizes these observations follows quite naturally:

```python
def partitions(elements: str, k: int) -> set:
    """Returns the list of k-partitions of elements.

    Each part is non-empty!

    Args:
        elements (str): set represented by a string with unique elements
        k (int): number of parts in a partition

    Returns:
        set[tuple[str, ...]]: k-tuples, each with non-empty components
    """
    if k == 1:  # the first base case
        return set([elements])
    if k == len(elements):  # the second base case
        return set([tuple(elements)])
    # the recursive argument!
    result = set()
    for a_partition in partitions(elements[1:], k-1):  # case 1
        lst = [elements[0]] + list(a_partition)
        result.add(tuple(lst))
    for a_partition in partitions(elements[1:], k):  # case 2
        for i in range(k):
            lst = list(a_partition)
            lst[i] = elements[0] + lst[i]
            result.add(tuple(lst))
    return result
```

## 0.4 Code Testing

Whether a piece of code runs correctly or not, is based on the **specifications** for the code. There are essentially three kinds of testing regimens:

- **Blackbox** testing: develop the tests based purely on the specifications. The tests, often called unit tests, can be constructed even before the code is written!

We will see an example of black-box testing for our `partitions` function.

- **Inline** testing: Unlike traditional testing that is looking for some sort of input-output consistency, inline testing relies on being able to construct **invariants** associated with the **state** of the program on a generic input.
- **Whitebox** testing: This is the kind of testing associated with trying to **review** code for correctness. As a side-effect, it helps us determine where complexity in the code can be simplified, based on the *flow-of-control* through the code on various inputs.

In a large project with multiple functional parts, unit-testing is first applied to individual functions. More robust black-box testing can help identify errors that are caused by **integration** of various independently developed functions.

### 0.4.1 Example: Blackbox testing

The simplest way to set up tests is via `assert` statements where the code checks whether a condition that **ought to be true** is indeed true as implemented!

Let's look at the specification of the `partitions` function above to try and identify some tests that, if successful on a particular implementation of the function, would give us some confidence in the correctness of the implementation. Here is the docstring:

```
partitions(elements: str, k: int) -> set
    Returns the list of k-partitions of elements.

    Each part is non-empty!

    Args:
        elements (str): set represented by a string with unique elements
        k (int): number of parts in a partition

    Returns:
        set[tuple[str, ...]]: k-tuples, each with non-empty components
```

1. Test the **boundary** case when `k` equals 1

```
# All five elements in one part of the 1-partition
partns1 = partitions('abcde', 1)
assert len(partns1) == 1
assert len(partns1.pop()) == 5
```

2. Test the **boundary** case when the size of `elements` equals `k`:

```
# Every element in its own part of the unique 5-partition
partns2 = partitions('abcde', 5)
assert len(partns2) == 1
letters = tuple('abcde')
```

```
solo_partition = partns2.pop()
for letter in tuple('abcde'):
    assert letter in solo_partition
```

3. Consider a case that can be easily analyzed, e.g., all partitions of `abcde` into 2 parts. Any such partition will have of course have 2 non-empty parts, and all the elements must be in one part or the other but not both. This translates to ensuring that the sum of the lengths of the parts in every partition equals 5:

```
partns = partitions('abcde', 2)
assert all(len(part1 + part2) == 5 for part1, part2 in partns)
```

4. Furthermore, a simple combinatorial analysis can be used to **count** the number of such partitions of `abcde` into 2 parts. Choosing one part automatically fixes the other part (remember that order is not important). So the only possibilities are for one part to have a single element and the other have 4, or one part to have 2 elements and the other have 3. Hence, the number of partitions is:

$$\binom{5}{1} + \binom{5}{2} = 5 + 10 = 15$$

```
partns = partitions('abcde', 2)
assert len(partns) == 15
```

### 0.4.2 Test Infrastructure

Recall that there are two ways to organize code in modules:

- create definitions in the module so that the module (and its namespace) can be imported and used elsewhere, or
- create a stand-alone **script** that can be executed directly by the interpreter.

Blackbox testing is meant for importable modules, not scripts.

#### 0.4.2.1 Testing using the "main" block

We can add an auxiliary **main** block at the end of the module to test the module contents:

```
if "__name__" == "__main__":
    <code for testing definitions in the module>
```

In this block, we can either:

- use **doctests** via the `doctest` module. Note that these tests are often very limited and depend heavily on the exact string representation of the result.
- use `assert` statements as shown in the example above.

#### 0.4.2.2 Test function suites

This is the preferred and more robust mechanism to test a target module, and the tests are typically organized as follows:

```
target_module.py
tests
  test_fn1.py
  test_fn2.py
  ...
```

In this example, the `test_fn*.py` modules contain suites of tests meant to test various parts of `target_module.py`; each discrete test is generally implemented by a small function. We will use the third-party `pytest` package in such a test framework. Please install the package via `pip` on your laptops for next week!