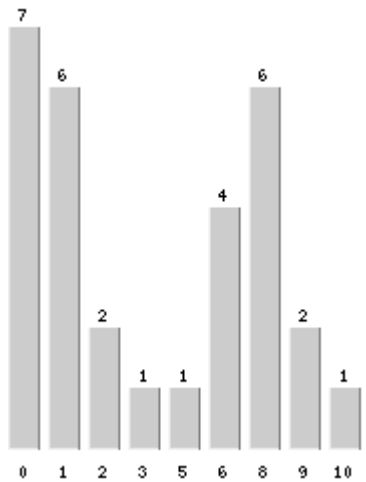
Lecture 6 – Modules, Namespaces

https://cw.fel.cvut.cz/wiki/courses/be5b33prg/start

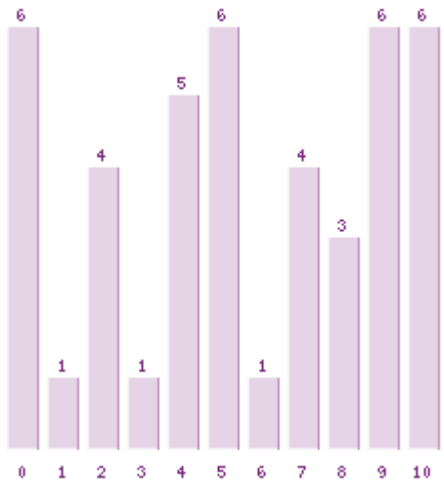
Milan Nemy

Czech Technical University in Prague,
Faculty of Electrical Engineering, Dept. of Cybernetics
https://beat.ciirc.cvut.cz/people/milan-nemy/
milan.nemy@cvut.cz

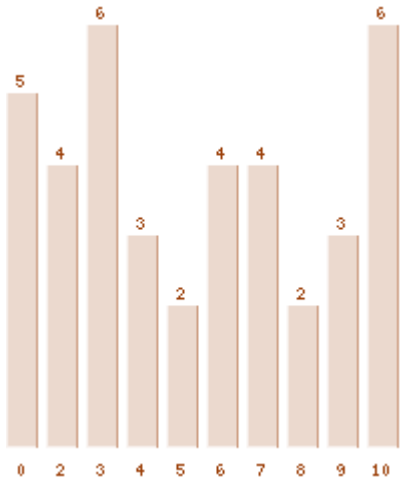
MIDTERM TEST EVALUATION



2021/2022



2022/2023



2023/2024

- Solutions uploaded: 39
- Average score: 5.08
- Median score: 5

source http://openbookproject.net/thinkcs/python/english3e/dictionaries.html

RECAP: DICTIONARIES

```
>>> letter_counts = {}
>>> for letter in "Mississippi":
...     letter_counts[letter] = letter_counts.get(letter, 0) + 1
...
>>> letter_counts
{'M': 1, 's': 4, 'p': 2, 'i': 4}
```

- EXAMPLE: *Function that **counts the number of occurrences** of a letter in a string using a frequency table of the letters in the string (how many times each letter appears)*
- Dictionary ideal for frequency tables

source http://openbookproject.net/thinkcs/python/english3e/dictionaries.html

MODULES – INTRODUCTION

- A **module** is a **file** containing Python definitions and statements intended for use in other Python programs

- Modules are one of the main **abstraction layers** available and probably the most natural one
- Abstraction layers allow **separating code** into parts holding **related data** and **functionality**
- The most natural way to separate these two layers is to regroup **all interfacing functionality into one file**, and all **low-level operations in another file**

- Loading modules done with the **import** and **from … import** statements

source http://openbookproject.net/thinkcs/python/english3e/modules.html

NAMESPACES

- Namespace is a **mapping** *from names to objects*, i.e. space where given name exists …
- Namespace is a **collection of identifiers** that *belong to a module, function, or a class*
- Namespace is **set of symbols** used to **organize objects** of various kinds so that we can *refer to them by name*
- Namespaces allow programmers to work on the same project without having **naming collisions** (*allow name reuse*)

source http://openbookproject.net/thinkcs/python/english3e/modules.html

NAMESPACES

- Namespaces are often **hierarchically** structured (*good practice!*)
- Each name must be **unique in its namespace**
- Namespace is very general concept not really limited only to Python
- Each **module has its own namespace**, i.e. *we can use the same identifier name in multiple modules without causing an identification problem*

source http://openbookproject.net/thinkcs/python/english3e/modules.html

How are namespaces defined in Python?

- **Packages** (*collections of related modules*)
- **Modules** *(.py files containing definitions of functions, classes, variables, etc.)*
- **Classes, Functions** …

source http://openbookproject.net/thinkcs/python/english3e/modules.html

What is the difference between programs and modules?

- Both are stored in **.py files**.
- **Programs** (*scripts*) are designed to be run (*executed*)
- **Modules** (*libraries*) are designed to be *imported* and used by other programs and other modules
- **Special case**: *.py file* is designed to be both a program and a module, i.e. it can be executed as well as imported to provide functionality for other modules

source http://openbookproject.net/thinkcs/python/english3e/modules.html

MODULES – NAMESPACES – FILES

- Python has a convenient and simplifying *one-to-one* mapping: **one module per file** giving rise to **one namespace**
- Python takes the **module name from the filename**, and this becomes the **name of the namespace**

- EXAMPLE: *math.py* is a **filename**, the module is called **math**, and its namespace is **math**
  (*in Python the concepts are more or less interchangeable*)

In other languages (e.g. C#) one module can span multiple files, or one file to have multiple namespaces, or many files to all share the same namespace

source http://openbookproject.net/thinkcs/python/english3e/modules.html

Where does Python look for imported modules?

- A list named **path** in module **sys** contains all the locations (*and this list can be modified*)
- The **first item** on the list is the **directory** with the program

source http://openbookproject.net/thinkcs/python/english3e/modules.html

What happens when Python imports a module?

- Python *searches* the module among already imported modules in **sys.modules**
  (If the module is *not found* in **sys.modules**, Python searches locations in **sys.path**)

- Python then *executes* the module once it is found,
- and *records* that the module has already been imported
- Python *creates names in local* **namespace** for the imported module, or for all the variables, functions, etc. imported from the module

source http://openbookproject.net/thinkcs/python/english3e/modules.html

MODULES – RUNNING & IMPORTING


```
if __name__ == "__main__":
    print("This won't run if I'm  imported.")
```

- Modules that can be both **run** and **imported**
- Special variable called **__name__** inside each module it contains:
  - the **name of the module** (of the *.py* file) when the module is imported
  - string **"__main__"** when the *.py* file is run as a program (script)
- Variable **__name__** is defined in both the calling namespace and in the namespace of the module

source http://openbookproject.net/thinkcs/python/english3e/modules.html

MODULES – RUNNING & IMPORTING

Very bad

```
[...]
from modu import *
[...]
x = sqrt(4)  # Is sqrt part of modu? A builtin? Defined above?
```

Better

```
from modu import sqrt
[...]
x = sqrt(4)  # sqrt may be part of modu, if not redefined in between
```

Best

```
import modu
[...]
x = modu.sqrt(4)  # sqrt is visibly part of modu's namespace
```

- To gain access to symbols defined in a module (i.e. in a different namespace) module has to be imported (3 ways)
- The statements **load a module**, **create a name** in the current namespace, and **bind the name to the loaded module**

source http://docs.python-guide.org/en/latest/writing/structure/#modules

MODULES – RANDOM NUMBERS


```python
import random

# Create a black box object that generates random numbers
rng = random.Random()

dice_throw = rng.randrange(1,7)   # Return an int, one of 1,2,3,4,5,6
delay_in_seconds = rng.random() * 5.0
```

- To include **random decision-making process**
- To take **samples from probability distributions**
- To play a game of chance where the computer needs to throw some dice, pick a number, or flip a coin …
- To shuffle a deck of playing cards randomly …
- In **modelling and simulations**: weather models, environmental models, Monte Carlo method
- To encrypt banking sessions on the internet …

source http://openbookproject.net/thinkcs/python/english3e/modules.html

MODULES – RANDOM NUMBERS


```python
import random

# Create a black box object that generates random numbers
rng = random.Random()

dice_throw = rng.randrange(1,7)   # Return an int, one of 1,2,3,4,5,6
delay_in_seconds = rng.random() * 5.0
```

- The **random** method returns a *floating point number* in the interval [0.0, 1.0) — *the square bracket means "closed interval on the left" and the round parenthesis means "open interval on the right"* – 0.0 is possible, but all returned numbers will be *strictly less than* 1.0.
- It is usual to **scale the results** after calling this method to get them into an interval suitable for application.
- EXAMPLE: scaling to a number in the interval [0.0, 5.0) *(uniformly distributed numbers — numbers close to 0 are just as likely to occur as numbers close to 0.5 or close to 1.0)*

source http://openbookproject.net/thinkcs/python/english3e/modules.html

MODULES – RANDOM NUMBERS


```
r_odd = rng.randrange(1, 100, 2)
```

- EXAMPLE: *We needed a random odd number less than 100*
- The **randrange** method generates an integer between its **lower** and **upper** argument
- The **randrange** method has same semantics as range (so the *lower bound is included*, but the *upper bound is excluded*)
- All the values have an **equal probability** of occurring (i.e. *the results are uniformly distributed*)
- Randrange also takes an optional **step argument** (like range)

source http://openbookproject.net/thinkcs/python/english3e/modules.html

MODULES – RANDOM NUMBERS

```
cards = list(range(52))  # Generate ints [0 .. 51]
                         #    representing a pack of cards.
rng.shuffle(cards)       # Shuffle the pack
```

This example shows how to **shuffle** a list.
(*shuffle cannot work directly with range object so list type converter first is necessary*)

```
drng = random.Random(123)  # Create generator with known starting state
```

- Random number generators are based on a deterministic algorithm — **repeatable** and **predictable**
- Called **pseudo-random generators** (*not genuinely random*)

- Each time you ask for another random number, you'll get one based on the current **seed attribute**, and the **state of the seed**

source http://openbookproject.net/thinkcs/python/english3e/modules.html

MODULES – RANDOM NUMBERS

```
drng = random.Random(123)  # Create generator with known starting state
```

- Repeatability for **debugging** and for writing **unit tests** *(programs that do the same thing every time they are run)*
- Forcing the random number generator to be initialized with a **known seed** every time (*often this is only wanted during testing / back-testing*)
- Without this seed argument, the system probably uses something based on the OS time …

source http://openbookproject.net/thinkcs/python/english3e/modules.html

MODULES – RANDOM NUMBERS

```python
import random

def make_random_ints(num, lower_bound, upper_bound):
    """
      Generate a list containing num random ints between lower_bound
      and upper_bound. upper_bound is an open bound.
    """
    rng = random.Random()  # Create a random number generator
    result = []
    for i in range(num):
        result.append(rng.randrange(lower_bound, upper_bound))
    return result
```

```python
>>> make_random_ints(5, 1, 13)  # Pick 5 random month numbers
[8, 1, 8, 5, 6]
```

- EXAMPLE: generate a list containing *given number of random generated integers between a lower and upper bound*
- NOTE: that we got **duplicates** in the result (*often this is wanted, e.g. if we throw a die five times, we would expect some duplicates*)

source http://openbookproject.net/thinkcs/python/english3e/modules.html

How to take care of duplicates?

- If you wanted **5 distinct months**, then this algorithm is wrong
- In this case a good algorithm is to generate the list of possibilities, **shuffle** it, and **slice off** the number of elements you want:

```
xs = list(range(1,13))  # Make list 1..12  (there are no duplicates)
rng = random.Random()   # Make a random number generator
rng.shuffle(xs)         # Shuffle the list
result = xs[:5]         # Take the first five elements
```

source http://openbookproject.net/thinkcs/python/english3e/modules.html

MODULES – RANDOM NUMBERS

- Allowing **duplicates** (*usually described as pulling balls out of a bag with replacement*)
- No **duplicates** (*usually described as pulling balls out of the bag without replacement*)
- Algorithm **"shuffle and slice"** is not ideal for case of choosing few elements from a very large domain
  (*Assume we need five numbers between 1 and 10 million, without duplicates. Generating a list of ten million items, shuffling it, and then slicing off the first five would be a performance disaster*)

source http://openbookproject.net/thinkcs/python/english3e/modules.html

MODULES – RANDOM NUMBERS

```python
import random

def make_random_ints_no_dups(num, lower_bound, upper_bound):
    """
      Generate a list containing num random ints between
      lower_bound and upper_bound. upper_bound is an open bound.
      The result list cannot contain duplicates.
    """
    result = []
    rng = random.Random()
    for i in range(num):
        while True:
            candidate = rng.randrange(lower_bound, upper_bound)
            if candidate not in result:
                break
        result.append(candidate)
    return result

xs = make_random_ints_no_dups(5, 1, 10000000)
print(xs)
```

This agreeably produces 5 random numbers, without duplicates:

```
[3344629, 1735163, 9433892, 1081511, 4923270]
```

Choose wisely the algorithm based on the input data!

source http://openbookproject.net/thinkcs/python/english3e/modules.html

MODULES – TIME

- consstant = $\mathcal{O}(1)$
- logarithmic = $\mathcal{O}(\log n)$
- linear = $\mathcal{O}(n)$
- n log n = $\mathcal{O}(n \log n)$
- quadratic = $\mathcal{O}(n^2)$
- cubic = $\mathcal{O}(n^3)$
- polynomial = $\mathcal{O}(n^k)$
- exponential = $\mathcal{O}(k^n)$
- factorial =$\mathcal{O}(n!)$

How efficient and reliable is our code?

- One way to experiment is to time **how long** various operations take and what the **memory requirements** are *related to algorithm complexity:* *https://people.duke.edu/~ccc14/sta-663/AlgorithmicComplexity.html*

- The **time** module has a function **clock** that is recommended
- Whenever clock is called, it returns a **floating point number** representing *how many seconds have elapsed since your program started running (varies according to the OS type!)*

source http://openbookproject.net/thinkcs/python/english3e/modules.html

MODULES – TIME

```python
import time

def do_my_sum(xs):
    sum = 0
    for v in xs:
        sum += v
    return sum

sz = 10000000        # Lets have 10 million elements in the list
testdata = range(sz)

t0 = time.clock()
my_result = do_my_sum(testdata)
t1 = time.clock()
print("my_result    = {0} (time taken = {1:.4f} seconds)"
        .format(my_result, t1-t0))

t2 = time.clock()
their_result = sum(testdata)
t3 = time.clock()
print("their_result = {0} (time taken = {1:.4f} seconds)"
        .format(their_result, t3-t2))
```

```
my_sum    = 49999995000000 (time taken = 1.5567 seconds)
their_sum = 49999995000000 (time taken = 0.9897 seconds)
```

- <u>EXAMPLE</u>: *Generating and summing up ten million elements in under a second*
- Proprietary function runs **57% slower** than the **built-in** one.

source http://openbookproject.net/thinkcs/python/english3e/modules.html

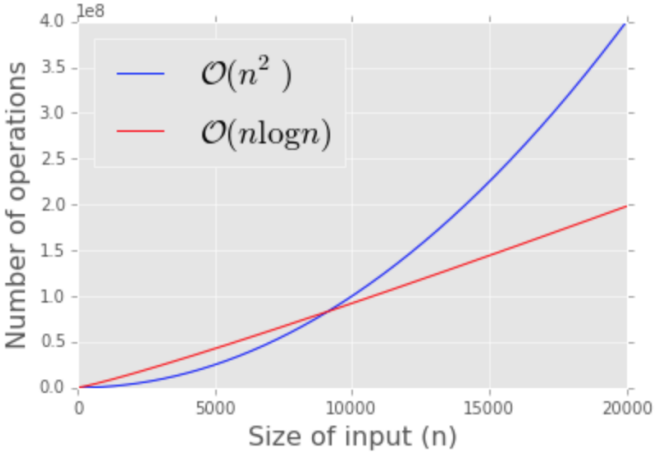
Comparing complexity of $\mathcal{O}(n^2)$ (e.g. bubble sort) and $\mathcal{O}(n \log n)$ (e.g. merge sort).

```
def f1(n, k):
    return k*n*n

def f2(n, k):
    return k*n*np.log(n)

n = np.arange(0, 20001)

plt.plot(n, f1(n, 1), c='blue')
plt.plot(n, f2(n, 1000), c='red')
plt.xlabel('Size of input (n)', fontsize=16)
plt.ylabel('Number of operations', fontsize=16)
plt.legend(['$\mathcal{O}(n^2)$', '$\mathcal{O}(n \log n)$'], loc='best', fontsize=20);
```




```
# Notice how much overhead Python objects have
# A raw integer should be 64 bits or 8 bytes only

print sys.getsizeof(1)
print sys.getsizeof(1234567890123456789012345678901234567890)
print sys.getsizeof(3.14)
print sys.getsizeof(3j)
print sys.getsizeof('a')
print sys.getsizeof('hello world')
```

```
24
44
24
32
38
48
```

source https://people.duke.edu/~ccc14/sta-663/AlgorithmicComplexity.html

MODULES – MATH


```
>>> import math
>>> math.pi                    # Constant pi
3.141592653589793
>>> math.e                     # Constant natural log base
2.718281828459045
>>> math.sqrt(2.0)             # Square root function
1.4142135623730951
>>> math.radians(90)           # Convert 90 degrees to radians
1.5707963267948966
>>> math.sin(math.radians(90))  # Find sin of 90 degrees
1.0
>>> math.asin(1.0) * 2         # Double the arcsin of 1.0 to get pi
3.141592653589793
```

- The **math** module contains mathematical functions typically found on a calculator, including mathematical constants like *pi* and *e*

- Functions **radians** and **degrees** to convert angles
- Mathematical functions are **pure** and do not have any state

source http://openbookproject.net/thinkcs/python/english3e/modules.html

MODULES – CREATING MODULES

```
def remove_at(pos, seq):
    return seq[:pos] + seq[pos+1:]
```

```
>>> import seqtools
>>> s = "A string!"
>>> seqtools.remove_at(4, s)
'A sting!'
```

- Create own **modules** – save script as a file with **.py extension**
- EXAMPLE: function *remove_at* in a script is saved as a file named *seqtools.py*
- The module must be first imported before the use (***.py*** *is the file extension and is not included in the import statement)*
- RECOMMENDATION: break up very large programs into manageable sized parts and keep related parts together

source http://openbookproject.net/thinkcs/python/english3e/modules.html

MODULES – NAMESPACES

```python
# Module1.py

question = "What is the meaning of Life, the Universe, and Everything?"
answer = 42
```

```python
# Module2.py

question = "What is your quest?"
answer = "To seek the holy grail."
```

```python
import module1
import module2

print(module1.question)
print(module2.question)
print(module1.answer)
print(module2.answer)
```

```
What is the meaning of Life, the Universe, and Everything?
What is your quest?
42
To seek the holy grail.
```

source http://openbookproject.net/thinkcs/python/english3e/modules.html

MODULES – NAMESPACES

```python
def f():
    n = 7
    print("printing n inside of f:", n)

def g():
    n = 42
    print("printing n inside of g:", n)

n = 11
print("printing n before calling f:", n)
f()
print("printing n after calling f:", n)
g()
print("printing n after calling g:", n)
```

```
printing n before calling f: 11
printing n inside of f: 7
printing n after calling f: 11
printing n inside of g: 42
printing n after calling g: 11
```

- Functions also have *own namespaces*
- Functions can read (**read-only**) variable in the outer **scope**

- EXAMPLE: *the three **n**'s above do not collide since they are each in a **different namespace** — three names for three different variables*

source http://openbookproject.net/thinkcs/python/english3e/modules.html

MODULES – NAMESPACES – FILES

- RECOMMENDATION: **keep the concepts distinct in your mind**

- **Files** and **directories** organize where code and data are stored
- **Namespaces** and **modules** are a **programming concepts**: *they help us organize how we want to group related functions and attributes*
- Namespaces are **not** about "where" to store things, and should not have to coincide with the file structures

If the file math.py is renamed, its module name needs to be changed, import statements need to be changed, and the code that refers to functions or attributes inside that namespace also needs to be changed accordingly – this leads to **refactoring**

source http://openbookproject.net/thinkcs/python/english3e/modules.html

MODULES – SCOPE

A **scope** is a *textual region of a Python program where a namespace is directly accessible*

What types of scopes can be defined?

- **Local scope** refers to identifiers declared within a function / class *(these identifiers are kept in the namespace that belongs to the function, and each function has its own namespace)*
- **Global scope** refers to all the identifiers declared within the current module, or file
- **Built-in scope** refers to all the identifiers built into Python *(those like range() and min() that can be used without having to import anything*)

source http://openbookproject.net/thinkcs/python/english3e/modules.html

MODULES – SCOPE

```
def range(n):
    return 123*n

print(range(10))
```

```
n = 10
m = 3
def f(n):
    m = 7
    return 2*n+m

print(f(5), n, m)
```

What are the scope precedence rules?

- The same name can occur in more than one of these scopes, but **the innermost, or local scope, will always take precedence** over the global scope, and the global scope always gets used in preference to the built-in scope
- Names can be "**hidden**" from use if own variables or functions reuse those names (we already discussed *shadowing*)
- EXAMPLE: *variables **n** and **m** are created just for the duration of the execution of **f** since they are created in the local namespace of function **f** (precedence rules apply)*

source http://openbookproject.net/thinkcs/python/english3e/modules.html

MODULES – THE DOT OPERATOR

```
import math
x = math.sqrt(10)
```

```
from math import cos, sin, sqrt
x = sqrt(10)
```

```
def area(radius):
    import math
    return math.pi * radius * radius

x = math.sqrt(10)      # This gives an error
```

```
from math import *    # Import all the identifiers from math,
                      #   adding them to the current namespace.
x = sqrt(10)          # Use them without qualification.
```

```
>>> import math as m
>>> m.pi
3.141592653589793
```

- Variables defined **inside a module** are called **attributes** of the module
- Attributes are accessed using the **dot** operator (**.**)
- When a dotted name is used it is often referred to it as a *fully qualified name*

source http://openbookproject.net/thinkcs/python/english3e/modules.html

REFERENCES

This lecture re-uses selected parts of the OPEN BOOK PROJECT
Learning with Python 3 (RLE)
http://openbookproject.net/thinkcs/python/english3e/index.html
available under GNU Free Documentation License Version 1.3)

- Version date: October 2012
- by Peter Wentworth, Jeffrey Elkner, Allen B. Downey, and Chris Meyers (based on 2nd edition by Jeffrey Elkner, Allen B. Downey, and Chris Meyers)
- Source repository is at https://code.launchpad.net/~thinkcspy-rle-team/thinkcspy/thinkcspy3-rle
- For offline use, download a zip file of the html or a pdf version from http://www.ict.ru.ac.za/Resources/cspw/thinkcspy3/