IMPULS
PRO KARIÉRU
A PRAXI

METODIKA ZÁKLADY PROGRAMOVÁNÍ

BLOKOVÉ PROGRAMOVÁNÍ PRO SŠ

EVROPSKÁ UNIE
Evropské strukturální a investiční fondy
Operační program Výzkum, vývoj a vzdělávání

MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY

Jhk.cz

JIHOČESKÁ HOSPODÁŘSKÁ KOMORA

Jihočeský kraj

Pracovní postup „Metodika Základy programování / blokové programování pro SŠ“ je součástí publikace „Pracovní postupy pro workshopy digitalizace ve školách.“, která vznikla v rámci aktivity Asistenčního centra Impuls pro kariéru a praxi při Jihočeské hospodářské komoře díky realizaci projektu „Implementace Krajského akčního plánu Jihočeského kraje III“, který je spolufinancován Evropskou unií. Registrační číslo projektu CZ.02. 3. 68/0.0/0.0/19_078/0018246

Elektronická verze publikace je k dispozici na **www.impulsprokarieru.cz**

Autor: Ing. Marta Vohnoutová
Editor: RNDr. Ing. Jana Kalová, Ph.D.
Publikaci připravila Přírodovědecká fakulta Jihočeské univerzity

Jihočeská univerzita v Českých Budějovicích
Přírodovědecká fakulta
Centrum pro přírodovědné vzdělávání

Grafický design: Čestmír Sukdol – www.brandi.cz

Vydala: Jihočeská hospodářská komora

2021

Obsah

Základní instrukce / **5**

Blokové programování / **5**

Poznámky / **6**

Začínáme / **6**

Algoritmy / **7**

UML / **9**

Algoritmus / **12**

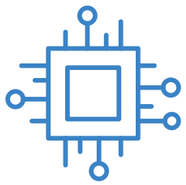
Základy Pythonu / **13**

Praktická část / **30**

Úlohy pro procvičování žáky / **35**

Grafy / **39**

Co jste se naučili / **46**



Cílem workshopu Metodika Základy programování/blokové programování je naučit posluchače "programátorsky" myslet. Začínáme základy algoritmizace, formulací problému, vytvořením algoritmu, nakreslením vývojového diagramu a konečně sestavením a odladěním programu. V každé části jsou vždy části teoretická a praktická, kde jsou jak příklady vyřešené učitelem, tak příklady pro posluchače k procvičení. Nakonec je zařazen vždy vzorový příklad, který by posluchači měli vyřešit samostatně nebo ve dvojici. Programovací jazyk zde není podstatný, pro svoji názornost a jednoduchost byl v praktických příkladech zvolen jazyk Python v.3.

Základní instrukce

Tento kurz je doporučován pro žáky středních škol. K tomu dokumentu je připraven materiál v prostředí Jupyterhub a/nebo Jupyter Notebook, kde si jednotlivé úlohy budou moci posluchači vyzkoušet. Toto prostředí bude posluchačům k dispozici odkudkoliv s internetovým připojením. Pokud budou chtít posluchači použít vlastní počítač, předpokládá učitel předinstalovaný základní programovací jazyk Python ve verzi 3.3 nebo vyšší.

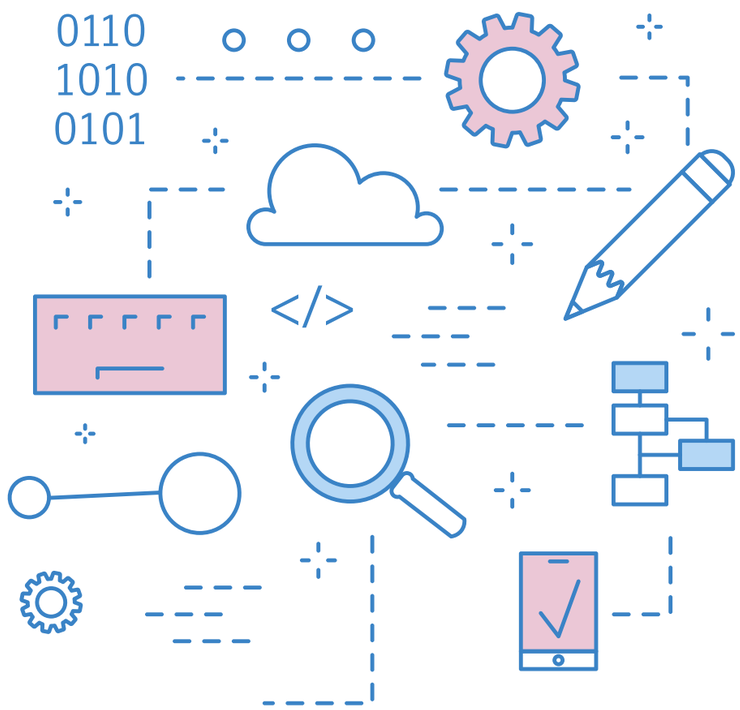
Znalost základů programování jakéhokoliv programovacího jazyka je žádoucí, nikoliv však nutná.

Časová dotace tohoto kurzu nemůže být zcela jednoznačná. Závisí na tom, zda se studenti již seznámili s programováním a zda již mají obecně nějaké zkušenosti s programováním, případně s Pythonem. Učitel by měl nejprve prostudovat všechny materiály dané k tomuto kurzu a pak se sám rozhodnout, kolik času kurzu věnovat.

Metodika používá jako zdroj informací

- Programátorskou cvičebnici Algoritmy v příkladech od Radka Pelánka
- Programátorské kuchařky MatfyzPress 2011
- Výuka algoritmizace na ZŠ – Bc. Jana Bromová
- http://www.ivt.mzf.cz/algoritmizace-a-programovani/uvod-do-algoritmu

Základní znalost programování není nutná, ale je žádoucí. Programovací jazyk není určen, v podstatě je možné použít jakýkoliv, v příkladech této metodiky byl použit jazyk Python pro jeho názornost.



Blokové programování

Blokové programování vychází z blokové struktury programu. To je v IT označení pro zdrojový kód programu rozčleněný do samostatných bloků, které ho rozdělují na souvislé logické části nebo samostatné funkční části u funkcí a procedur. Bloková struktura se hojně využívá hlavně v moderních technikách programování, které se označují také jako strukturované jazyky.

Poznámky

- Ačkoliv je v příkladech použit pro názornost jazyk Python, není toto výuka Pythonu.
- V učebnici, která byla zvolena jako základ najdete i obtížné kategorie, které v této metodice zásadně neuvádím.
- Pro žáky volte takové příklady, které složitostí odpovídají jejich možnostem a znalostem a které stihnou s přehledem udělat v určeném čase.

Začínáme

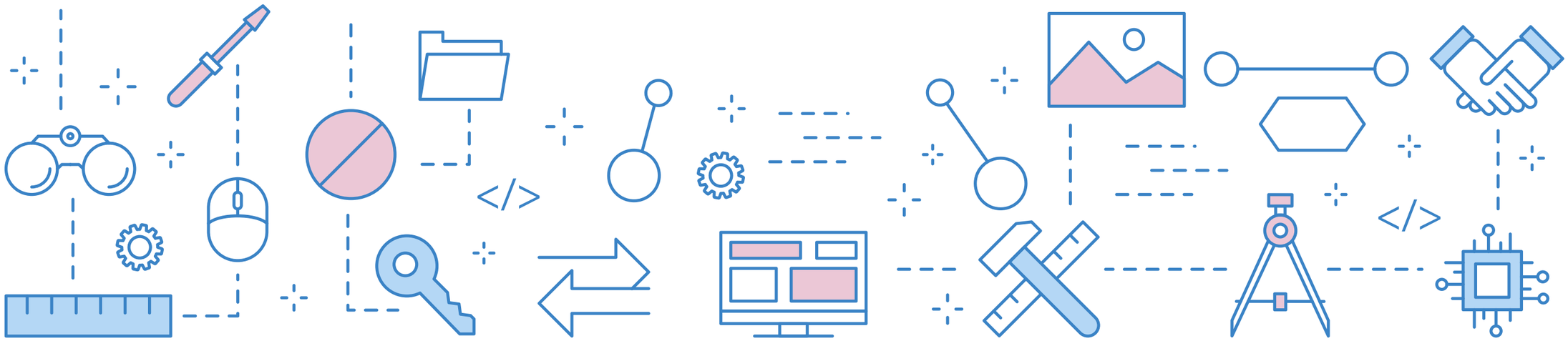
Kategorie obtížnosti

Kategorie obtížnosti jsou subjektivní.

V roli experta jste vůči žákům vy, vždy uvažujte s tímto poměrem

Nápad	Začátečník	Expert	Kódování	Začátečník	Expert
1	Vcelku jasné	Zřejmé	1	20 minut	5 minut
2	Trocha rozmýšlení	Rutina	2	1 hodina	Rutina
3	Těžké	Trocha rozmýšlení	3	Půl dne	1 hodina

„Mysl není nádoba, kterou je potřeba naplnit,
ale oheň, který je potřeba vznítit.”
Plutarchos



Algoritmy

Základní pojmy

Algoritmizace

Algoritmizace je přesný postup, který se používá při tvorbě programu pro počítač, jehož prostřednictvím lze řešit nějaký konkrétní problém.

Algoritmizaci lze rozdělit do několika kroků:

- Formulace problému
- Analýza úlohy
- Vytvoření algoritmu
- Sestavení programu
- Odladění programu

Formulace problému

V této etapě je třeba přesně formulovat požadavky, určit výchozí hodnoty, požadované výsledky, jejich formu a přesnost řešení. Tvůrce algoritmu musí dokonale rozumět řešenému problému, jinak nemůže algoritmus sestavit – v praxi programátoři spolupracují s odborníky z oblastí, pro které mají vytvořit algoritmus.

Analýza úlohy

Při analýze úlohy si ověříme, zda je úloha řešitelná a uděláme první návrh řešení. Definujeme způsob vstupu a tvar vstupních hodnot. Zjistíme, zda je úloha řešitelná a za jakých podmínek a zda má případně více řešení. Pokud má úloha více postupů, jak ji řešit (což je u složitějších úloh běžné), zvolíme jedno řešení, případně si zdůvodníme, proč právě toto řešení bylo zvoleno. Dále pak již postupujeme podle něj.

Vytvoření algoritmu úlohy

Algoritmus je přesný návod či postup, kterým lze vyřešit daný typ úlohy. Pojem algoritmu se nejčastěji objevuje při programování, kdy se jím myslí teoretický princip řešení problému (oproti přesnému zápisu v konkrétním programovacím jazyce). Obecně se ale algoritmus může objevit v jakémkoli jiném vědeckém odvětví.

Vývojový diagram (flowchart)

Vývojový diagram je druh diagramu, který slouží ke grafickému znázornění jednotlivých kroků algoritmu, pracovního postupu nebo nějakého procesu. Vývojový diagram obsahuje obrazce různého tvaru (obdélníky, kosočtverce, aj.), navzájem propojené pomocí šipek. Obrazce reprezentují jednotlivé kroky, šipky tok řízení. Vývojové diagramy standardně nezobrazují tok dat, ten je zobrazován pomocí data flow diagramů. Vývojové diagramy jsou často využívány v informatice během programování pro analýzu, návrh, dokumentaci nebo řízení procesu.

Tvorbě vývojových diagramů bude věnována jedna z dalších kapitolek.

Sestavení programu

Na základě algoritmu řešené úlohy sestavíme program (zdrojový text) v konkrétním programovacím jazyce. Ze zdrojového textu se pomocí překladače do strojového kódu vytvoří spustitelný program (případně interpretem se přeloží a spustí jednotlivé příkazy programu). Lze říci, že dobře provedená analýza úlohy a algoritmizace je velmi důležitá pro řešení daného problému a je základním předpokladem sestavení programu pro počítač.

Tvorbě programů v jazyce Python bude věnována jedna z dalších kapitolek.
Pro naši vzorovou ukázku **Ciferace** se použije tzv. rekursivní volání funkce, proto si jej znázorníme již nyní

Rekursivní volání funkce

Není to nic složitého, funkce prostě zavolá sama sebe. Ciferace je krásná ukázka rekurzivní funkce. Podmínkou je, že součet všech číslic v čísle musí být jednočíselný. Pokud není, dosadíme za číslo součet a zavoláme funkci znovu.

Odladění programu

Naprogramováním činnost nekončí. Program musí být tzv.“odblbený“. Odladěním chceme odstranit chyby z programu. Programátor musí spolu s testerem odchytit a ošetřit všechny možné, i velmi nepravděpodobné možnosti. Celá kapitola v programování je tzv.error handling tj. ošetřování možných chyb. Příkladem ošetření možné chyby je například zabránění dělení nulou, které by skončilo chybou.

Další odlaďování se týká např. toho, aby program běžel co nejrychleji a aby neměl velké nároky na paměť.

Nejčastější chyby jsou chyby v zápise, tzv. syntaktické – ty odhalí překladač a dělají je i zkušení programátoři. Horší jsou logické chyby, které vyplývají z nesprávně navrženého algoritmu – projeví se nesprávnou činností programu nebo špatnými výsledky – při odstraňování těchto chyb může pomoci ladící program (debugger) umožňující sledování aktuálního stavu proměnných a krokování. Teprve po odstranění všech druhů chyb můžeme program použít k praktickému řešení úloh. Důležité je následné testování. V týmu řešitelů jsou tzv.testeři.

V případě složitějších nebo komerčních úloh je tester role, která je oddělena od vývojáře, aby nevznikala profesní slepota. Testování a metodika testování je opět celá věda.

Algoritmizace v praxi

Proces psaní příkazů v programovacím jazyce se nazývá **programování**.

- **Fáze analýzy** – Co bude program umět definuje **analytik**, ten pak dává zadání pro **programátora (vývojáře)**.
- **Fáze programování** – Vývojář podle popisu vytvoří program (přepíše řešení do programovacího jazyka). Zápis programu v programovacím jazyce se nazývá **zdrojový kód**. Poté je spuštěn program, který dokáže přeložit tento zdrojový kód do jazyka počítače, a vznikne tak **spustitelný program**. Prvotní otestování programu se nazývá jeho **ladění (debugging)**.
- **Fáze testování** – Tento program dostane **tester**, který se snaží najít v programu chyby nebo nesprávné chování neodpovídající zadání. Pokud objeví chybu, pak ji předá zpět programátorovi k dořešení.
- **Fáze akceptace**, zaškolení obsluhy programu.
- **Fáze změnová, údržba a podpora** – Program je předán uživatelům. Dochází k požadavkům na úpravy nebo přidání funkčností.

UML

Unified Modeling Language (UML) je grafický modelovací jazyk pro specifikaci, vizualizaci a dokumentaci informačních systémů a aplikací. UML umožňuje popsat systém pomocí slov a obrázků. Můžeme jím modelovat různé systémy.

Výčet diagramů standardu UML 2.0:

Strukturní diagramy:

- Diagram tříd (Class Diagram)
- Diagram objektů (Object Diagram)
- Diagram komponent (Component Diagram)
- Diagram složených struktur (Composite Structure Diagram)
- Diagram nasazení (Deployment Diagram)
- Diagram balíčků (Package Diagram)

Diagramy chování:

- Diagram případů užití (Use Case Diagram)
- Diagram aktivit (Activity Diagram)
- Stavový diagram (State Machine Diagram)

Diagramy interakce:

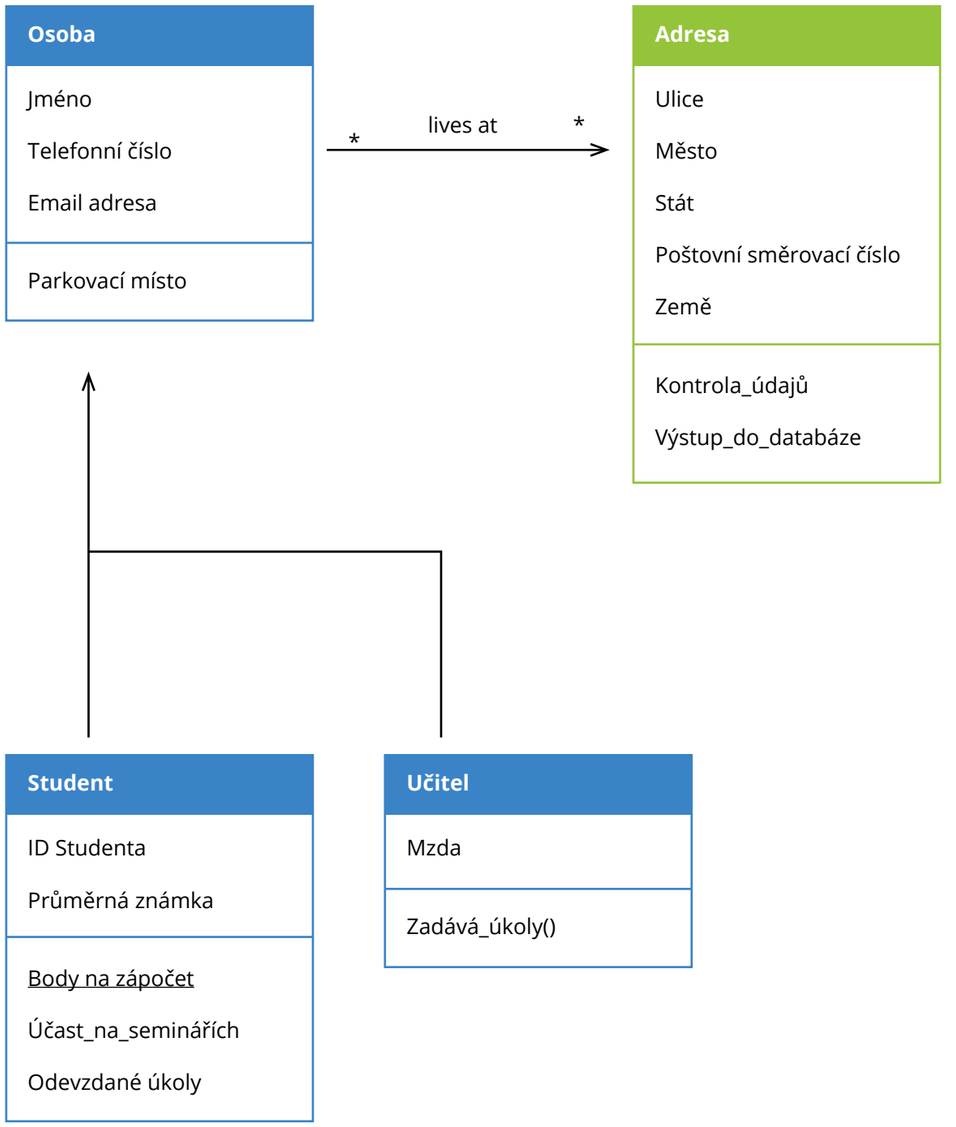
- Sekvenční diagram (Sequence Diagram)
- Diagram komunikace (Communication Diagram)
- Diagram přehledu interakcí (Interaction Overview Diagram)
- Diagram časování (Timing Diagram)

Často slyšíme pojem **diagramy tříd** (class diagrams) nebo **use-case diagramy**. UML je jazyk, který byl standardizován Object Management Group (OMG), mezinárodní asociací pro tzv."open standards for object-oriented applications (**http://www.omg.org**)."

Základy jazyka UML a schopnost se jazykem UML vyjádřit se naučí prakticky každý. Vyjadřováním prostřednictvím UML máme jistotu, že nám ostatní porozumí.

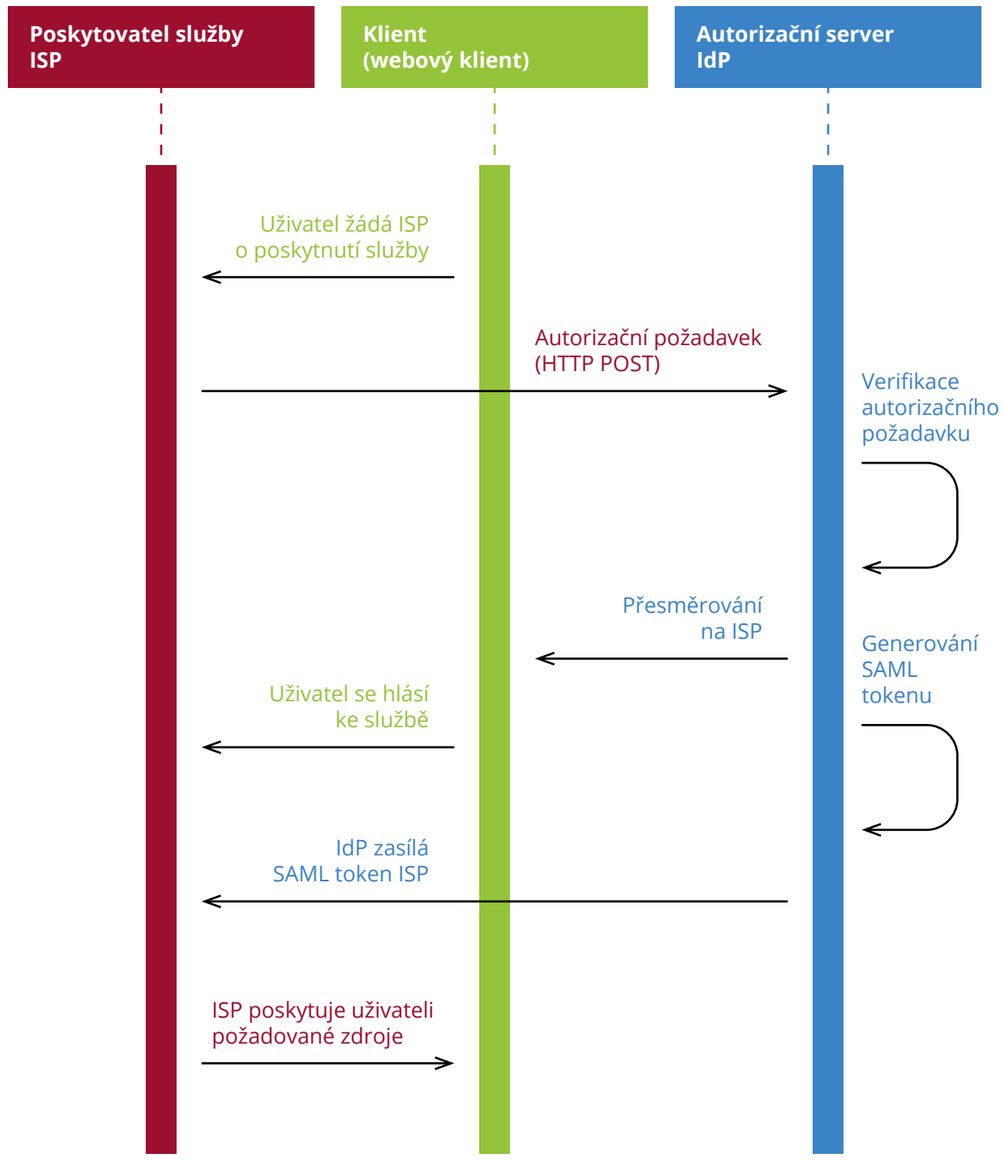
Aktuálně se používá UML verze 2.0, ("OMG Unified Modeling Language: Superstructure, Version 2.0, Revised Final Adopted Specification, October 2004", from **http://www.omg.org**).

Class diagram



UML Class diagram

Sekvenční diagram

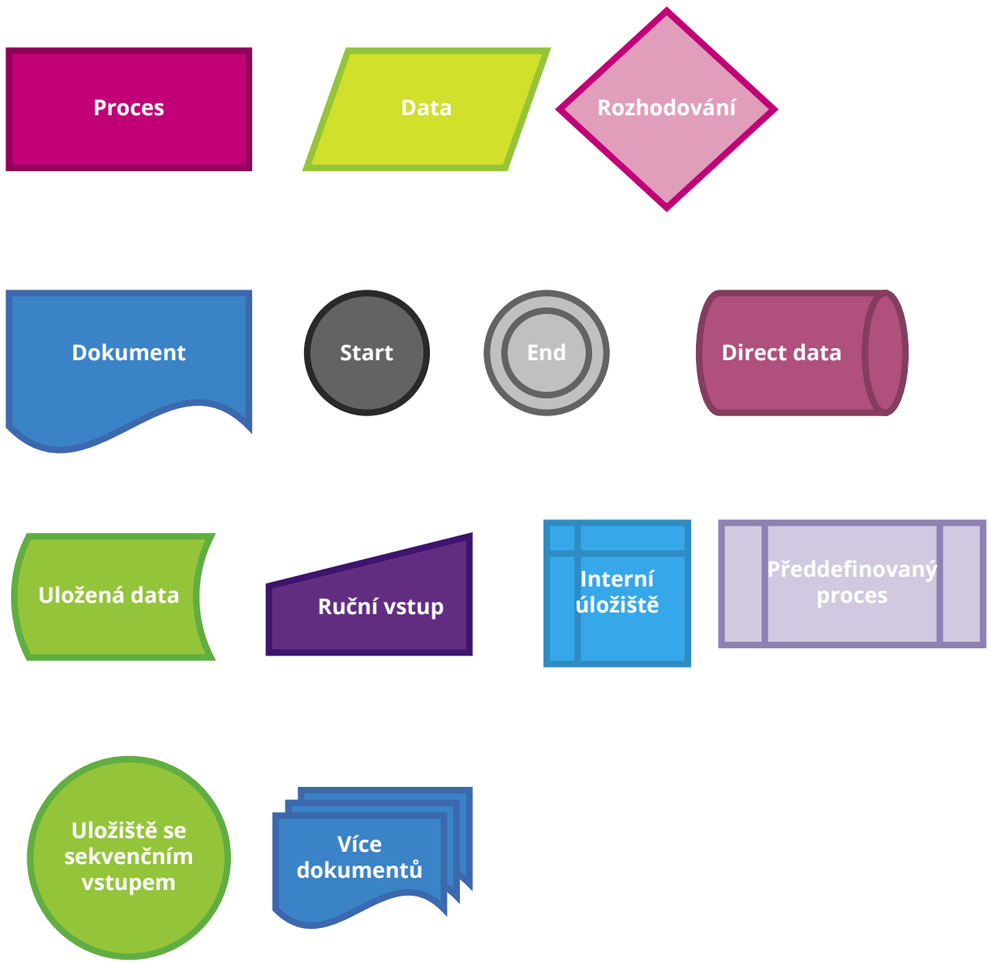


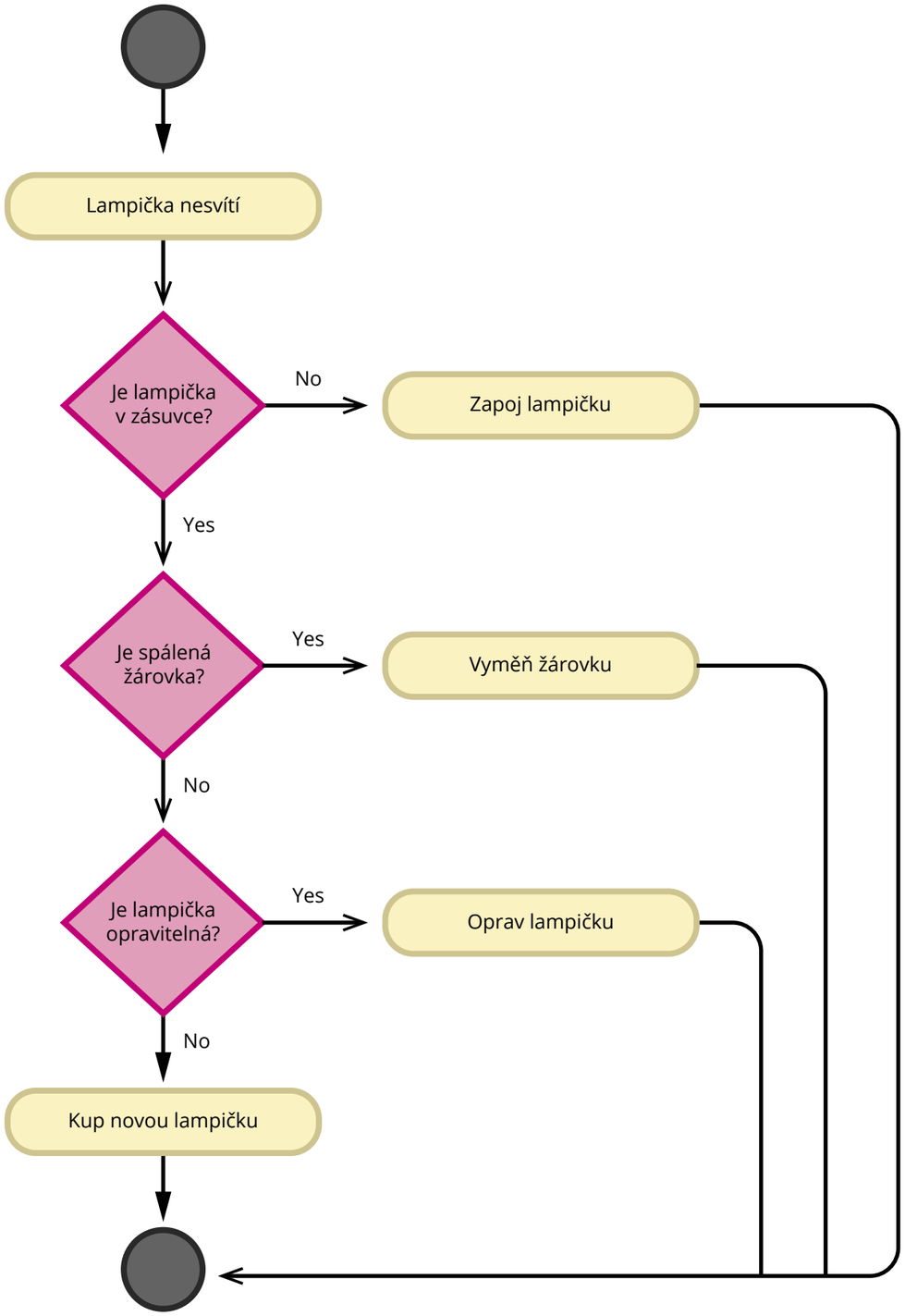
UML Sekvenční diagram – SAML protokol

Flowchart diagram

Více zde

Protože flowchart diagram (neboli vývojový diagram) se přímo týká programování, budeme u něj o něco podrobnější. Nejužívanější symboly vývojového diagramu jsou na následujícím obrázku.





Příklad vývojového diagramu je na následujícím obrázku. Zkusíme si však i některé vlastní

Algoritmus

Algoritmus je přesný návod či postup, kterým lze vyřešit daný typ úlohy. Příkladem je kuchařský recept.

Vlastnosti algoritmů

- **Elementárnost** – Algoritmus se skládá z konečného počtu jednoduchých (elementárních) kroků.
- **Konečnost** – Každý algoritmus musí skončit v konečném počtu kroků.
- **Obecnost** – Algoritmus neřeší jeden konkrétní problém (2/3), ale obecnou třídu obdobných problémů (a/b).
- **Determinovanost** – Algoritmus je determinovaný, pokud za stejných podmínek (pro stejné vstupy) poskytuje stejný výstup. Tohle je problematické - existují např.pravděpodobnostní algoritmy, kde do hry vstupuje náhoda.
- **Determinismus** – Každý krok algoritmu musí být jednoznačně a přesně definován; v každé situaci musí být naprosto zřejmé, co a jak se má provést, jak má provádění algoritmu pokračovat. Vyjádření výpočetní metody v programovacím jazyce se nazývá program.
- **Výstup** – Algoritmus vede od zpracování hodnot k výstupu.

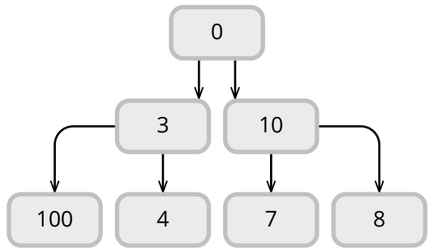
Metody návrhu algoritmů

Algoritmů je samozřejmě mnoho. V učebnicích se dočteme například o těchto:

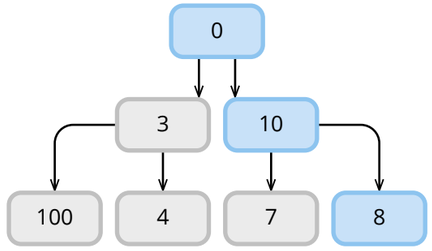
- Třídění
- Hladový algoritmus
- Halda
- Grafy
- Dijkstrův algoritmus
- Minimální kostra
- Rozděl a panuj
- Dynamické programování
- Vyhledávací stromy
- Hešování
- Řetězce a vyhledávání v textu
- Rovinné grafy
- Eulerovské tahy
- Toky v sítích
- Intervalové stromy

Pro praktickou ukázku si vybereme zjednodušený hladový algoritmus

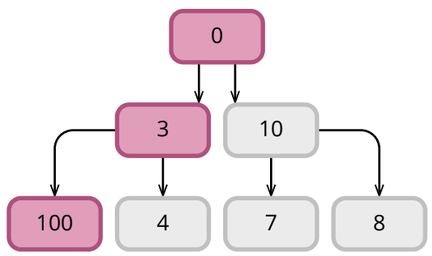
Stromový graf



Hladový algoritmus



Optimální algoritmus



Hladový algoritmus

Hladový algoritmus (anglicky greedy search) Takovými algoritmy rozumíme ty, které hledají řešení celé úlohy po jednotlivých krocích a splňují následující dvě podmínky:

- V každém kroku zvolí lokálně nejlepší řešení.
- Provedené rozhodnutí již nikdy neodvolává (tedy „nebacktrackuje“).

Lokálně nejlepší řešení je takové, které v aktuálním kroku vybere tu možnost, která nám na tomto místě nejvíce pomůže (bez jakéhokoliv ohledu na globální stav). Může to být třeba nejvyšší hodnota, nebo nejkratší cesta v grafu.

Pokud ale od hladového algoritmu chceme, aby nám našel **globálně nejlepší řešení**, musí naše úloha splnit předpoklad, že si výběrem lokálně nejlepšího řešení nezhoršíme to globální. Tento předpoklad se nedá formulovat obecně a je nutné se nad ním zamyslet zvlášť u každé úlohy.

Hladový algoritmus si představíme v Praktické části.

Základy Pythonu

Python je pravděpodobně nejsnazší současný programovací jazyk, s nímž se dobře pracuje, má obrovské množství knihoven téměř na cokoliv a velkou základnu, která vždy ráda pomůže. Kód v Pythonu se dobře píše i čte, proto jsem ho pro názorné ukázky algoritmizace a programování zvolila i já.

Více zde

Python je tzv. "cross-platform", což znamená, že běží na různých operačních systémech. Není potřeba nic kompilovat, což si za daň bere, že je o poznání pomalejší než např. jazyk C.

Budeme používat Python verze 3.3 a vyšší. Jak to zjistíme?
Prostě se optáme operačního systému, jaký Python je na něm nainstalován.

Jupyter Notebook

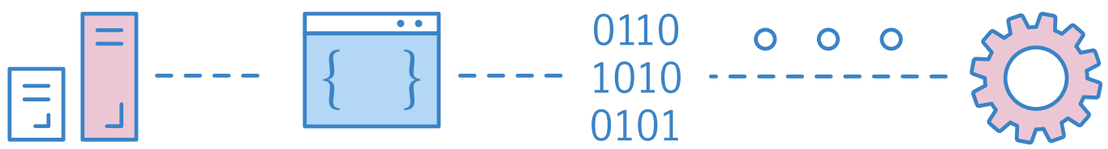
Příklady zde jsou uváděné v tzv. Jupyteru.

Poznámka: Pokud bude učitel ve výuce používat **Jupyterhub** nebo **Jupyter Notebook**, měl by jej v kurzu představit. V tomto materiálu však, vzhledem k rozsahu, uveden není.

```
!python -V
Python 3. 9. 1
```

Pokud nemáte Python nainstalován, zde je postup pro Linux Ubuntu.

```
sudo apt-get install python3
sudo apt-get install idle3
sudo apt-get install python3-pip
```



Jednoduché datové typy a vestavěné funkce v Pythonu

Vestavěné funkce

Python obsahuje řadu vestavěných funkcí, které nám mohou usnadnit život. Vidíte je v tabulce abecedně seřazené. **Podrobnější popis naleznete v QR kódu**.

abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	import()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

Jaké funkce a metody můžeme od třídy chtít, se dozvíme pomocí klíčového slova dir(). Zapamatujte si ho.

```
dir(„Marta“)

['capitalize',
 'casefold',
 'center',
 'count',
 'encode',
 'endswith',
 'expandtabs',
 'find',
 'format',
 'format_map',
 'index',
 'isalnum',
 'isalpha',
 'isascii',
 'isdecimal',
 'isdigit',
 'isidentifier',
 'islower',
 'isnumeric',
 'isprintable',
 'isspace',
 'istitle',
 'isupper',
 'join',
 'ljust',
 'lower',
 'lstrip',
 'maketrans',
 'partition',
 'removeprefix',
 'removesuffix',
 'replace',
 'rfind',
 'rindex',
 'rjust',
 'rpartition',
 'rsplit',
 'rstrip',
 'split',
 'splitlines',
 'startswith',
 'strip',
 'swapcase',
 'title',
 'translate',
 'upper',
 'zfill']
```

```
seasons = ['Spring', 'Summer', 'Fall', 'Winter']
dict(enumerate(seasons))
{0: 'Spring', 1: 'Summer', 2: 'Fall', 3: 'Winter'}compass.get_field_strength()
- intenzita magnetického pole
compass.heading() - azimut ve stupních, nutná inicializace pomocí compass.calibrate()
```


Základní datové typy

Každý programovací jazyk umí prezentovat položky dat. Python poskytuje více předdefinovaných datových typů, ale prozatím se budeme zabývat pouze několika z nich. Python představuje celá čísla (kladná a záporná celá čísla) pomocí typu int a řetězce (sekvence znaků Unicode) pomocí typu str. Zde je několik příkladů celých čísel a řetězců:

```
type(-973)
int
type(21062458333711437734086463779019080109822250862195507 2)
int
```

```
type('Tomáš Garrigue Masaryk')
str
type('αβγ ÷© + ▉ ┛◙☼Ş')
str
x='αβγ ÷© + ▉ ┛◙☼Ş'
len(x)
15
```

Příklad použití klíčového slova dir nám řekne, co můžeme od datového typu nebo funkce požadovat.

```
dir(x)
[….
 'capitalize',
 'casefold',
 'center',
 'count',
 'encode',
 'endswith',
 'expandtabs',
 'find',
….]
x.count(' ')
4
```

V textu jsou 4 mezery

Základní datové typy v Pythonu

Typ	Popis	Příklady hodnot
int	celá čísla	1, 42, -5, 200
float	reálná čísla (přesněji čísla v plovoucí desetinné čárce, přičemž Python používá desetinnou tečku, nikoliv čárku)	2.5, 3.25, -12.37832
bool	pravdivostní hodnoty	True, False
str	řetězce	"prase", "pes"

Základní datové typy Pythonu najdeme **v QR kódu**.

Složené datové typy

V Pythonu existuje pět základních složených datových struktur:

- bytearray
- bytes
- list
- str
- tuple

List

V Pythonu je list hlavní datovou strukturou používanou k ukládání sekvence prvků. Sekvence datových prvků uložených v seznamu nemusí být stejného typu.

Chcete-li vytvořit list, musí být datové prvky uzavřeny v [] a musí být odděleny čárkou. Následující kód například vytvoří dohromady čtyři datové prvky, které jsou různých typů.

```
můj_list = ["Rumcajs", 33,"Cipísek", True]
můj_list
['Rumcajs', 33, 'Cipísek', True]
type(můj_list)
list
```

List může obsahovat jiné listy i další jednoduché nebo složené datové typy.

```
můj_list =["Rumcajs",33,"Cipísek",True,[1,2,'Žluťoučký kůň úpěl ďábelské ódy']]
```


Slicing

s(-15)	s(-14)	s(-13)	s(-12)	s(-11)	s(-10)	s(-9)	s(-8)	s(-7)	s(-6)	s(-5)	s(-4)	s(-3)	s(-2)	s(-1)
T	**y**	**r**	**i**	**o**	**n**		**L**	**a**	**n**	**i**	**s**	**t**	**e**	**r**
s(0)	s(1)	s(2)	s(3)	s(4)	s(5)	s(6)	s(7)	s(8)	s(9)	s(10)	s(11)	s(12)	s(13)	s(14)

```
můj_list=list('Tyrion Lanister')

můj_list[:5] # od začátku 5 znaků
['T', 'y', 'r', 'i', 'o']

můj_list[:-1] # od konce všechny znaky
['r', 'e', 't', 's', 'i', 'n', 'a', 'L', ' ', 'n', 'o', 'i', 'r', 'y', 'T']

můj_list[:-3] # od konce každý třetí znak
['r', 's', 'a', 'n', 'r']

barvy=['Červená','Zelená','Modrá','Žlutá']
barvy[:2]
['Červená', 'Zelená']

barvy[-2]
'Modrá'

barvy[:2]
['Červená', 'Modrá']

barvy[:-2]
['Žlutá', 'Zelená']
```

```
s="I have a cat "
```

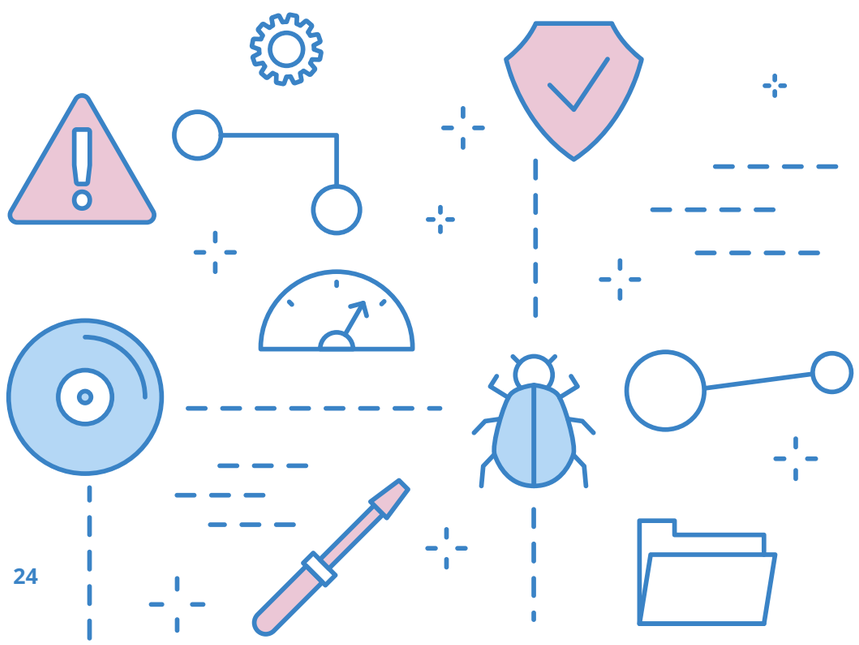
```
for i in s:
 print(s)
 s=s.lstrip(i)

I have a cat
```

```
 have a cat
have a cat
ave a cat
ve a cat
e a cat
 a cat
a cat
 cat
cat
at
t
```

Nesting

List podporuje vnořování (nesting), jednotlivé datové typy je možné uvnitř listu vnořovat.

```
a = [1,2,[100,200,300],['Marta',['Rudolf']],6]
max(a[2])
300


a[3][1][0]
'Rudolf'
```



Iterace

```
barvy
['Červená', 'Zelená', 'Modrá', 'Žlutá']
```

```
for barva in barvy:
 print(f'{barva} je v našem seznamu')

Červená je v našem seznamu
Zelená je v našem seznamu
Modrá je v našem seznamu
Žlutá je v našem seznamu
```

Lambda

Lambda je zvláštní typ funkce v Pythonu. Často se používá ve výpočtech ve složených datových typech. V příkladech si ukážeme časté použití lambda funkce.

Filtrování dat

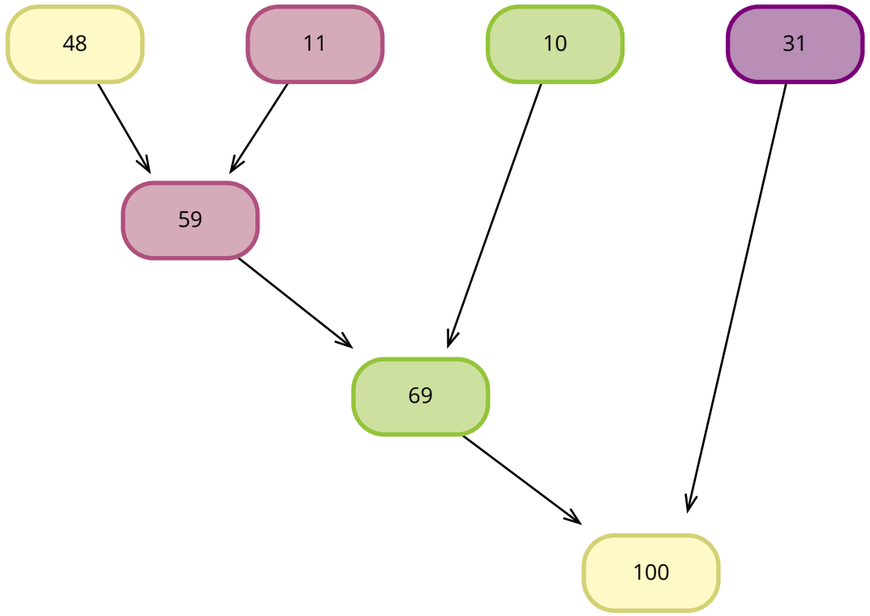
```
list(filter(lambda x: x > 100, [-5, 200, 300, -10, 10, 1000]))
[200, 300, 1000]


list(filter(lambda x: len(x)>5, barvy))
['Červená', 'Zelená']
```

Transformace dat

Pro transformaci dat využijeme ještě další funkci *map()*.

```
list(map(lambda x: x ** 2, [11, 22, 33, 44,55]))


[121, 484, 1089, 1936, 3025]


list(map(lambda x: f'{x} je delší než 5' if len(x)>5 else f'{x} je menší než 5',
barvy))


['Červená je delší než 5',
 'Zelená je delší než 5',
 'Modrá je menší než 5',
 'Žlutá je menší než 5']
```

Reduce - Agregace dat



Lambda – redukce

```
from functools import reduce #pro použití funkce reduce se musí importovat knihovna functools nebo její část
čísla=[48,11,10,31]
reduce(lambda x, y:x+y, čísla)
100
```


List comprehention

Netýká se pouze datové struktury list ale i dalších. Tato konstrukce bývá v Pythonu hojně využívána.

```
[i for i in barvy if len(i)>5]
['Červená', 'Zelená']

import math
čísla=[2,6,8,10,7,5]
[math.sqrt(x) for x in čísla]
[1.4142135623730951,
 2.449489742783178,
 2.8284271247461903,
 3.1622776601683795,
 2.6457513110645907,
 2.23606797749979]
```

List metody

Syntax	Description
L.append(x)	Přidává položku na konec listu L
L.count(x)	Vrátí počet výskytů položky x v listu L
L.extend(m)	L += m přidává všechny položky na konec listu L
L.index(x, start, end)	Vrací index pozice hledané položky, pokud je jich víc, tak první zleva; pokud položku nenajde odpoví ValueError exception
L.insert(i, x)	Vkládá položku x do listu L na pozici int i
L.pop()	Vrací hodnotu položky a zároveň odstraňuje výskyt položky v listu L. Pokud je položek více, odstraní první zprava
L.pop(i)	Vrací hodnotu položky a zároveň odstraňuje výskyt položky v listu L na pozici int i v listu L
L.remove(x)	Odstraní výskyt položky x v listu L, pokud položku x nenajde odpoví ValueError exception
L.reverse()	Obrátí pořadí položek v listu L
L.sort(...)	Setřídí list L

```
barvy.append(['Oranžová',"Duhová"])
barvy.extend(['Fialová',"Purpurová"])

barvy

['Červená',
 'Zelená',
 'Modrá',
 'Žlutá',
 ['Oranžová', 'Duhová'],
 'Fialová',
 'Purpurová']
```

Range

```
for i in range(2,25,3): # rozsah od 2(včetně) do 25(ne včetně) krok jsou 3
 print(i, end=(","))
2,5,8,11,14,17,20,23,
```

Tuples

Datová struktura, která je immutable - tedy pouze pro čtení. Oproti listu má velmi málo možností - count a index. Tuply jde sčítat.

```
t= (1,2,3,'Marta')

t += ("Vohnoutová",)

type(t)

tuple

dir(t)

[…
 'count',
 'index']
```


Dictionary (slovník)

Slovník je datová struktura, která je tvořena vždy klíčem (key) a hodnotou (value). Klíč musí být v dictionary jedinečný. V Pythonu se dictionary využívá velmi často. Uzavírá se mezi složené závorky {}

```
barvy_semaforu ={
 "Stop": "Červená",
 "Připrav se": "Oranžová",
 "Jeď": "Zelená"
}

barvy_semaforu.get("Stop")
'Červená'

barvy_semaforu['Stop']
'Červená'

barvy_semaforu['Barva co není v semaforu'] = 'Růžová' # přidám do slovníku další položku
```

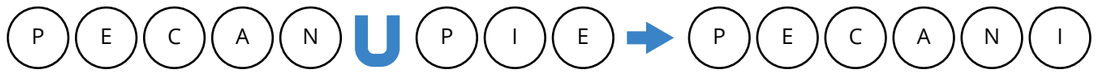
```
barvy_semaforu
{'Stop': 'Červená',
 'Připrav se': 'Oranžová',
 'Jeď': 'Zelená',
 'Barva co není v semaforu': 'Růžová'}

barvy_semaforu.keys()

dict_keys(['Stop', 'Připrav se', 'Jeď', 'Barva co není v semaforu'])

barvy_semaforu.values()
dict_values(['Červená', 'Oranžová', 'Zelená', 'Růžová'])

barvy_semaforu.items()
dict_items([('Stop', 'Červená'), ('Připrav se', 'Oranžová'), ('Jeď', 'Zelená'),
('Barva co není v semaforu', 'Růžová')])
```

Set (sada)

Typ set je datový typ, který podporuje některé množinové operace a je iterovatelný. Stejně jako dictionary se set uvozuje složenými závorkami {}. Set neřeší pořadí položek.

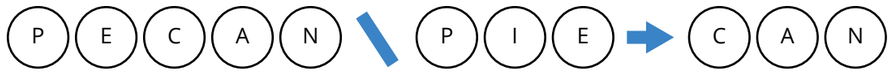
```
zelené = {"lupení","tráva","volno na semaforu"}
červené = {"rudá růže","meloun","stop na semaforu"}
žluté = {"lupení","tráva","pampelišky","sluníčko"}

type(zelené)
set

dir(zelené)
[…
 'add',
 'clear',
 'copy',
 'difference',
 'difference_update',
 'discard',
 'intersection',
 'intersection_update',
 'isdisjoint',
 'issubset',
 'issuperset',
 'pop',
 'remove',
 'symmetric_difference',
 'symmetric_difference_update',
 'union',
 'update']
```


Set - množinové operace

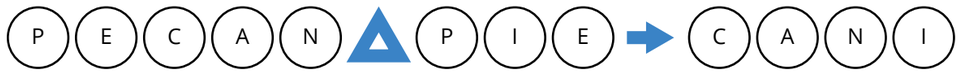


set(“PECAN”)|set(“PIE”) –> {‘A’, ‘C’, ‘E’, ‘I’, ‘N’, ‘P’} # sjednoceni

set(“PECAN”) & set(“PIE”) –> {‘P’, ‘E’} # průnik



set(“PECAN”) - set(“PIE”) –> {‘C’, ‘A’, ‘N’} # rozdíl



set(“PECAN”) ^ set(“PIE”) –> {‘C’, ‘A’, ‘N’, ‘I’} # symetrický rozdíl

```
set("PECAN")|set("PIE") # sjednocení
{'A', 'C', 'E', 'I', 'N', 'P'}

zelené | žluté

{'lupení', 'pampelišky', 'sluníčko', 'tráva', 'volno na semaforu'}
```

Aritmetické operace

Více zde

1. Aritmetické operace
2. Srovnávací operace
3. Přiřazovací operace
4. Logické operace
5. Bitové operace
6. Membership operace
7. Identitní operace

Aritmetické operace

1. sčítání +
2. odčítání -
3. násobení *
4. dělení /
5. umocňování **

```
8**3 # umocňování
512
```

Srovnávací operace

```
a, b=2,6
a <= b, a!= b, a >= b, a > b
(True, True, False, False)
```

Přiřazovací operace

```
a=5
b=a
```

Logické operace

1. and
2. or
3. not

```
a, b=2,6
a <= b and a!= b and a >= b and a > b
```



```
False
```

Membership operace

```
"Zelená" in barvy
True
```

Identitní operace

```
a=[12.10,7]
b=[12.10,7]
a == b #Srovnávací operace
True
a is b #Identitní operace
False
```

Flow control

Zpracovávají cykly a podmínky.

1. if-elif-else
2. while
3. for

while

```
while True:
 num = input("Vložte celé kladné číslo nebo nulu pro konec: ")
 try:
 num = int(num)
 except ValueError:
 num = -1
 if num < 0:
 print ("Prosím pouze celé kladné číslo.")
 continue
 elif num == 0:
 print ("Končím program…")
 break
 num_cube = (num ** 3)
```

```
 print ("Krychle o straně {0} má objem {1}.".format (num, num_cube))
print ("Nashledanou….")

Vložte celé kladné číslo nebo nulu pro konec: a
Prosím pouze celé kladné číslo.
Vložte celé kladné číslo nebo nulu pro konec: -5
Prosím pouze celé kladné číslo.
Vložte celé kladné číslo nebo nulu pro konec: 88
Krychle o straně 88 má objem 681472.
Vložte celé kladné číslo nebo nulu pro konec: 0
Končím program…
Konec programu.
```

break continue pass

```
print("break")
for i in range(0,10):
 if i == 6:
 print('here',end=',')
 break
 print(i, end=',')
print('\n')
print("continue")
for i in range(0,10):
 if i == 6:
 print("here",end=',')
 continue
 print(i, end=',')
print('\n')

print("pass")
for i in range(0,10):
 if i == 6:
 print("here",end=',')
 pass
 print(i, end=',')
print('\n')
break
0,1,2,3,4,5,here,
```



```
continue
0,1,2,3,4,5,here,7,8,9,
pass
0,1,2,3,4,5,here,6,7,8,9,
```

If-elif-else

```
t=(100000,999,9999)
for lines in t:
 if lines < 1000:
 print(lines," small")
 elif lines < 10000:
 print(lines," medium")
 else:
 print(lines," large")
100000 large
999 small
9999 medium
```

Vytváření a volání funkcí

def jméno_funkce(argumenty):

tělo funkce

```
def vyber_sudá_čísla(seznam_čísel):
 return [i for i in seznam_čísel if i%2 == 0]

print(vyber_sudá_čísla([4,7,88,21,15,4,72,62,112,332,331]))
[4, 88, 4, 72, 62, 112, 332]

def vyber_dělitelná_čísla(seznam_čísel, dělitel):
 return [i for i in seznam_čísel if i%dělitel == 0]

print(vyber_dělitelná_čísla([4,9,88,21,15,4,72,62,112,332,333,331,12],3))
[9, 21, 15, 72, 333, 12]
```

Praktická část

Praktická část předpokládá u každého příkladu postupovat v krocích, co jsme se naučili:

1. Formulace problému
2. Analýza úlohy
3. Vytvoření algoritmu úlohy - vývojový diagram
4. Sestavení programu a jeho odladění

Jako příklad si vezměme jednoduchou úlohu, často používanou ve hře geocaching – ciferaci.

Ciferace - 1.úloha prováděná učitelem

Ciferace - formulace problému

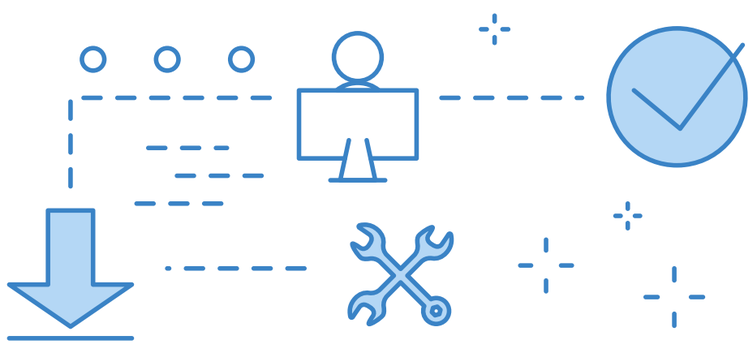
Při ciferaci sečteme všechny číslice z daného čísla. Pokud výsledný součet není jednociferný, součet opakujeme do té doby, až dostaneme jednociferné číslo. Toto jednociferné číslo je výsledkem ciferace.

Ciferace – analýza úlohy

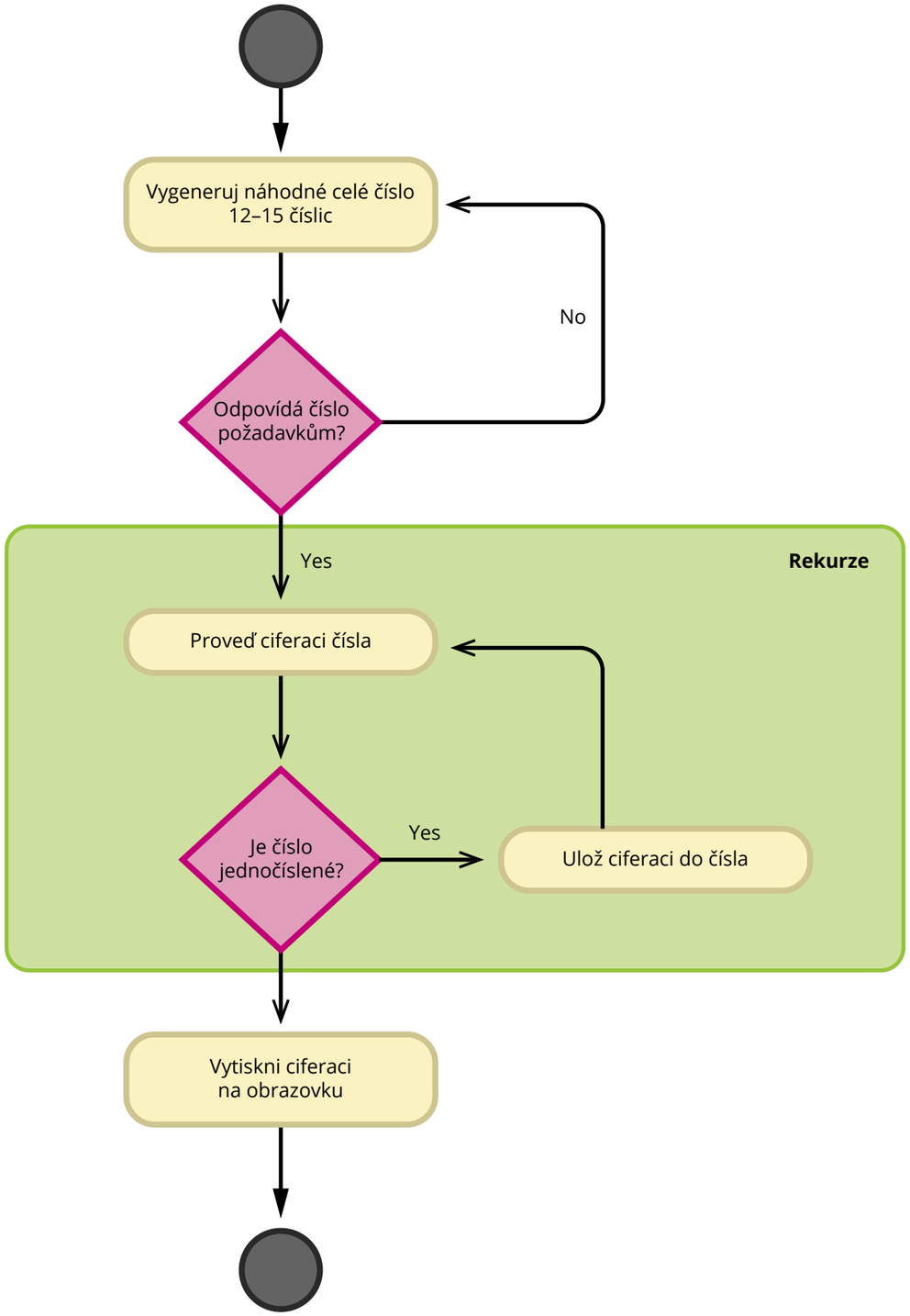
Zadání čísla – náhodné číslo o délce 15 až 20 číslic vybere počítač.

Zkontrolujeme, zda číslo odpovídá našim požadavkům:

– Číslo je celé kladné číslo
– Číslo obsahuje 15 až 20 číslic



Ciferace – vytvoření algoritmu úlohy – vývojový diagram



Cyklus

Rozdělte číslo na x číslic

Sečtěte x číslic

Zbyla vám jediná číslice?

– Pokud ano, vytiskněte ji na obrazovku a program končí.

– Pokud ne, uložte součet do čísla a vraťte se na Cyklus.

Ciferace - sestavení programu

```
#ciferace
#Autor: Marta Vohnoutová
```

```
import random as rd

def ciferace(číslo):
 c=sum([int(i) for i in str(číslo)])
 if len(str(c))==1:
 return c
 else:
 return ciferace(c)

while True:
 číslo = rd.randint(1e12,1e16)
 if isinstance(číslo, int) and číslo in range(int(1e12), int(1e16)):
 break


číslo = 248877668265883584
print(f'Pro číslo {číslo} je ciferace {ciferace(číslo)}')
Pro číslo 248877668265883584 je ciferace 6
```


Najděte slova podle zadaného vzoru – 2.úloha prováděná učitelem

Slova podle vzoru - formulace problému

Vytvořte Python program, který:

1. Stáhněte si slovník dictionary.txt, který najdete v **QR kódu**.
2. Ve slovníku dictionary.txt najděte všechna slova odpovídající zadanému vzoru:

Zadaný vzor je 0. 1. 2. 3. 2. 4. 5. 6. 7 Slova odpovídající zadanému vzoru musí splňovat - příklad pro vzor ’0. 0. 1. 2. 3’ [’AARON’, ’LLOYD’, ’OOZED’]

Slova podle vzoru - analýza úlohy

Zadání vzoru ve tvaru - list, uvnitř čísla oddělená čárkami např.[0,1, 1, 2, 3,1]

Zkontrolujeme, zda vzor odpovídá našim požadavkům:

– list
– uvnitř čísla oddělená čárkami

Stáhni a otevři soubor dictionary.txt

Cyklus

Ze souboru čti slovo po slovu

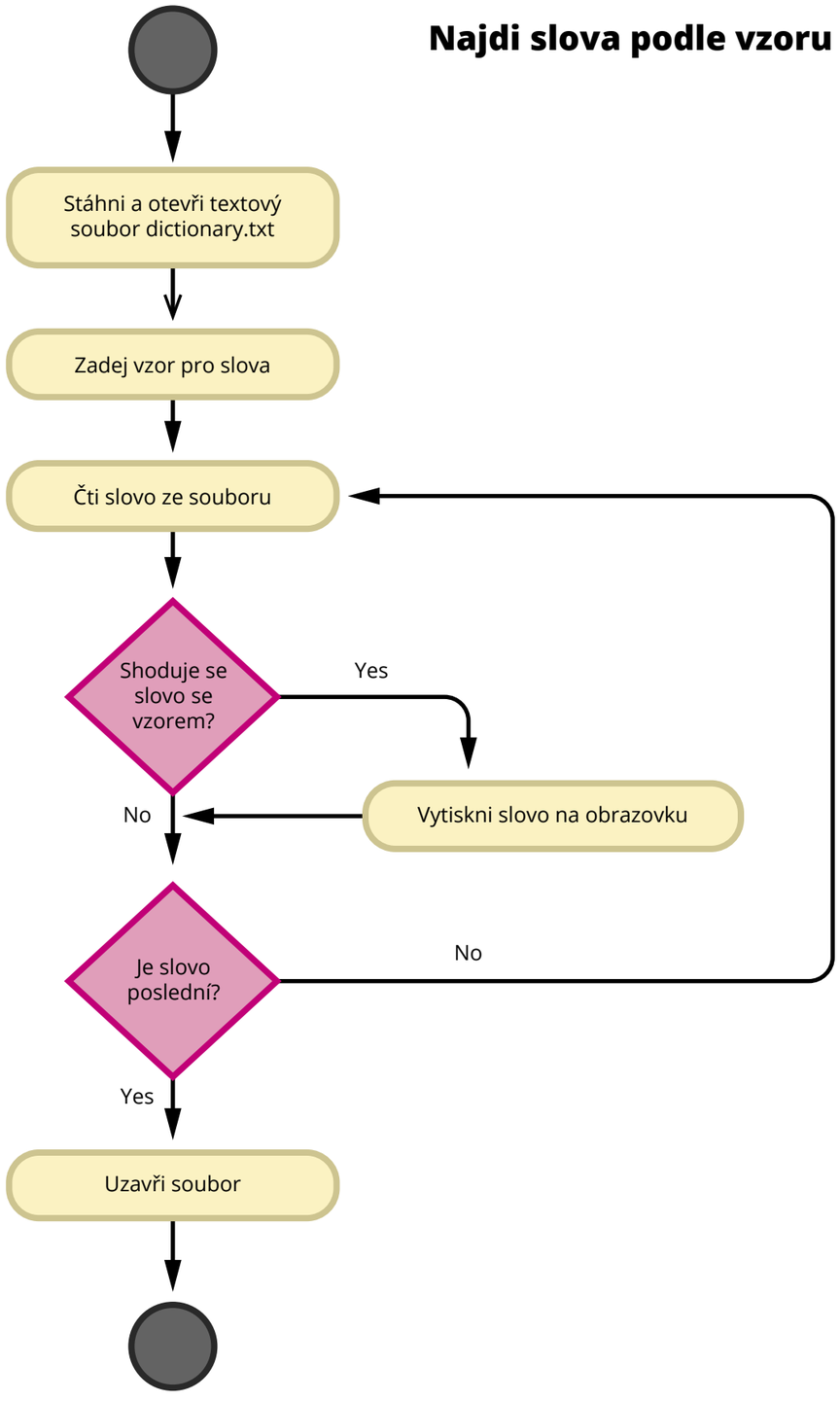
Zavolej funkci, která zjistí, zda slovo odpovídá vzoru

– Pokud ano, vytiskněte slovo na obrazovku

Je slovo poslední?

– Pokud ano, uzavřete soubor a program končí
– Pokud ne, přečtěte další slovo

Slova podle vzoru - vývojový diagram



Slova podle vzoru - sestavení programu

```
# Marta make patterns find words
import urllib
urllib.request.urlretrieve("https://raw.githubusercontent.com/MartaVohnoutovaBukovec/IOS-655-Python-a-Bash/master/dictionary.txt", "/home/marta/Zakladni_skoly_2021/dictionary.txt")


f = open("/home/marta/Zakladni_skoly_2021/dictionary.txt","r")
pattern_to_find=[0,1, 1, 2, 3,1]


def make_pattern(word):

 w=""
 pattern=[]
 for i in range(len(word)):
 if w.find(word[i]) == -1:
 w += word[i]
 pattern.append(w.find(word[i]))

 return pattern == pattern_to_find

print(pattern_to_find)
for word in f:
 if make_pattern(word.rstrip('\n')):
 print(word, end='')

f.close()

[0, 1, 1, 2, 3, 1]
ADDEND
ASSETS
ATTEST
BEETLE
EGGING
ESSAYS
FEEBLE
ISSUES
NEEDLE
SEETHE
```

Úlohy pro procvičování žáky

Úlohy pro procvičování žáky nemají přiložené řešení, ale při školení učitelů je možné řešení učitelům předat. U každé úlohy je označená její obtížnost, která je samozřejmě relativní. Protože se jedná především o výuku algoritmizace a programového myšlení, tak, ačkoliv úlohy nejsou nijak složité, je vyžadováno provedení všech kroků:

1. Formulace problému
2. Analýza úlohy
3. Vytvoření algoritmu úlohy - vývojový diagram
4. Sestavení programu a jeho odladění

Úloha č.1 – snadná – Matematické hádanky

Napište Python program, který vyřeší následující matematické hádanky:

1. Najděte správná celá čísla A, B, C, D, E, F, pro která platí ABCDEF × 3 = BCDEFA, kde ABCDEF je celé kladné číslo, které se vytvoří spojením jednotlivých číslic A, B, C, D, E, F. Správné možnosti vytiskněte jako *list* a v něm *tuply* takto: [(A1, B1, C1, D1, E1, F1), (A2, B2, C2, D2, E2, F2)...].
   Příklad výstupu: [(1,4,2,8,5,7), (2,8,5,7,1,4)]
2. Najděte celé kladné číslice označené L, O, G, I, C, pro které bude platit (L+O+G+I+C)3 = LOGIC, kde LOGIC je celé kladné číslo vytvořené sdružením (concatenating) číslic L, O, G, I, C. Výsledek vytiskněte na obrazovku jako list obsahující tuply takto: [(L1, O1, G1, I1, C1), (L2, O2, G2, I2, C2)...].
3. Najděte celé kladné číslice označené C, O, W, E, D, pro které bude platit COW × COW = DEDCOW, kde COW vznikne sdružením jednotlivých číslic C, O, W. Výsledek vytiskněte na obrazovku jako list obsahující tuply takto: [(C1, O1, W1, E1, D1), (C2, O2, W2, E2, D2)...].

Důležitá poznámka: Číslice mohou být v rozsahu 0...9 a v řešení se nesmí opakovat.

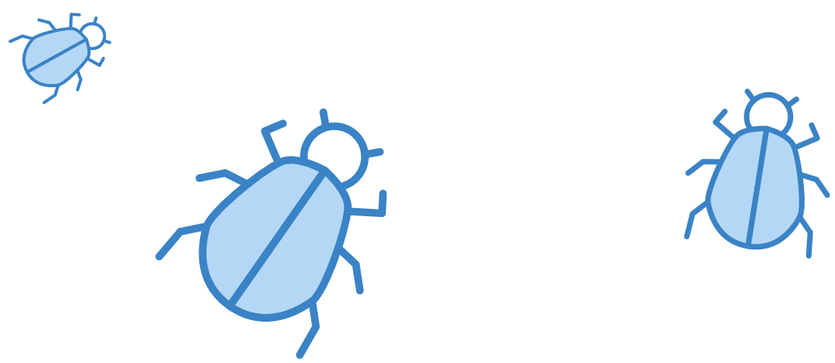
Nápověda: Použijte permutace k vytvoření možných kombinací čísel. Permutace vám zpřístupní import

```
from itertools import permutations as pm
```

```
# Marta Vohnoutová - úkol1 - Matematické hádanky
from itertools import permutations as pm

def ukol1_1():
 vystup=[]
 r=range(0,10)
 x=list(pm(r,6))
```



```
 for i in x:
 if (i[0]*100000+i[1]*10000+i[2]*1000+i[3]*100+i[4]*10+i[5]) *3 == (i[1]*100000+i[2]*10000+i[3]*1000+i[4]*100+i[5]*10+i[0]):
 vystup.append(i)
 print('Ukol 1.1',vystup)


def ukol1_2():
 vystup=[]
 r=range(0,10)
 x=list(pm(r,5))
 for i in x:
 if (i[0]+i[1]+i[2]+i[3]+i[4])**3 == (i[0]*10000+i[1]*1000+i[2]*100+i[3]*10+i[4]):
 vystup.append(i)
 print('Ukol 1.2',vystup)


def ukol1_3():
 vystup=[]
 r=range(0,10)
 x=list(pm(r,5))
 for i in x:
 if (i[0]*100+i[1]*10+i[2])*(i[0]*100+i[1]*10+i[2]) == (i[4]*100000+i[3]*10000+i[4]*1000+i[0]*100+i[1]*10+i[2]):
 vystup.append(i)
 print('Ukol 1.3', vystup)


ukol1_1()
ukol1_2()
ukol1_3()
Ukol 1.1 [(1, 4, 2, 8, 5, 7), (2, 8, 5, 7, 1, 4)]
Ukol 1.2 [(0, 4, 9, 1, 3), (0, 5, 8, 3, 2), (1, 9, 6, 8, 3)]
Ukol 1.3 [(3, 7, 6, 4, 1)]
```



Úloha č.2 – snadná – Cyklus

Pro 10 náhodných čísel v rozsahu od 19 do 999 a pro číslo 27 vytvořte cyklus, který bude provádět následující kroky:

1. Jestliže je číslo n sudé, proveďte operaci n/2
2. Jestliže je číslo n liché, proveďte operaci 3n + 1
3. celý cyklus opakujte, dokud nebude číslo n == 1
4. Pro každé číslo počítejte počet kroků než dosáhne číslo n hodnoty 1.

Vytiskněte všechny kroky pro každé číslo n a vytiskněte číslo s největším počtem kroků.

Nápověda: Náhodné číslo v daném rozsahu dostanete takto:

```
from random import randint as rd
n=rd(19,999)
```

```
# Marta Vohnoutová - úkol2 - Cyklus

from random import randint as rd

def ukol2():
 vystup={}
 for _ in range(0,10):
 vystup[rd(19,999)]=[]

 vystup[27]=[]

 for k, v in vystup.items():
 n=k
 while True:
 vystup[k].append(n)
 if n==1:
 break
 if n%2 == 0: # sude
 n=n/2
 elif n%2!= 0: # liche
 n=3*n+1
 return vystup
```



```
vystup=ukol2()
#print(vystup)
cislo=0
steps=0
for k, v in vystup.items():
 if len(v)>steps:
 steps=len(v)
 cislo=k
print(f'Pro číslo {cislo} je maximum {steps} kroků k jedničce')

Pro číslo 27 je maximum 112 kroků k jedničce
```

Úloha č.3 – snadné – Počítání samohlásek

Napište program v Pythonu, který spočte počet samohlásek ('a', 'e', 'i', 'o', 'u','y') v každém slově.

Zjednodušení: Slova jsou oddělena vždy mezerami, písmena jsou všechna malá a bez diakritiky. Např. věta „ucíme se python kazdy den“ bude mít výstup:

Výstup: Počet samohlásek je 1 1 1 3 2 2

```
# Marta Vohnoutová - úkol2 - Cyklus

def ukol3(veta):
 vowels = ('a', 'e', 'i', 'o', 'u','y')
 slova = veta.split()
 for i in slova:
 print(i, sum([i.count(j) for j in vowels]), end=',')

ukol3('ucime se python kazdy den')
ucime 3,se 1,python 2,kazdy 2,den 1,
```

Úloha č.4 – snadné – Veliká písmena

Z obrazovky zadejte nebo náhodně vygenerujte celé kladné číslo o několika číslicích. Na obrazovku pak vytiskněte číslo složené z velikých číslic. Abyste se netrápili s zadáním velikých číslic, tady je. Kdo si troufá, může hvězdičky zaměnit za konkrétní číslici.

Nápověda: Definování velkých číslic

```
Zero = [" *** ",
        " *   * ",
        "*     *",
        "*     *",
        "*     *",
        " *   * ",
        " *** "]
One = [" * ", "** ", " * ", " * ", " * ", " * ", "***"]
Two = [" *** ", "*   *", "* * ", " * ", " *   ", "*   ", "*****"]
Three = [" *** ", "*   *", "   *", " ** ", "   *", "*   *", " *** "]
Four = ["   * ", " ** ", " * * ", "* * ", "******", "   * ","   * "]
Five = ["*****", "*   ", "*   ", " *** ", "   *", "*   *", " *** "]
Six = [" *** ", "*   ", "*   ", "**** ", "*   *", "*   *", " *** "]
Seven = ["*****", "   *", "   * ", " * ", " *   ", "*   ", "*   "]
Eight = [" *** ", "*   *", "*   *", " *** ", "*   *", "*   *", " *** "]
Nine = [" ****", "*   *", "*   *", " ****", "   *", "   *", "   *"]

Digits = [Zero, One, Two, Three, Four, Five, Six, Seven, Eight, Nine]
```

Grafy

Více zde

Nahlédneme malinko do jednoho z nástrojů na vytváření grafů – jmenuje se **networkx**. Pokud ho chcete používat, musíte jej nejprve nainstalovat.

Instalace se provádí např. pomocí instalačního programu pip nebo conda.

```
pip install networkx
```

Dále se musí do prostředí Pythonu naimportovat. Pro náš příklad použijeme následující importy. Pokud v kurzu použijete připravené virtuální stroje, bude prostředí již připraveno, jinak si je musíte doinstalovat sami.

Importy

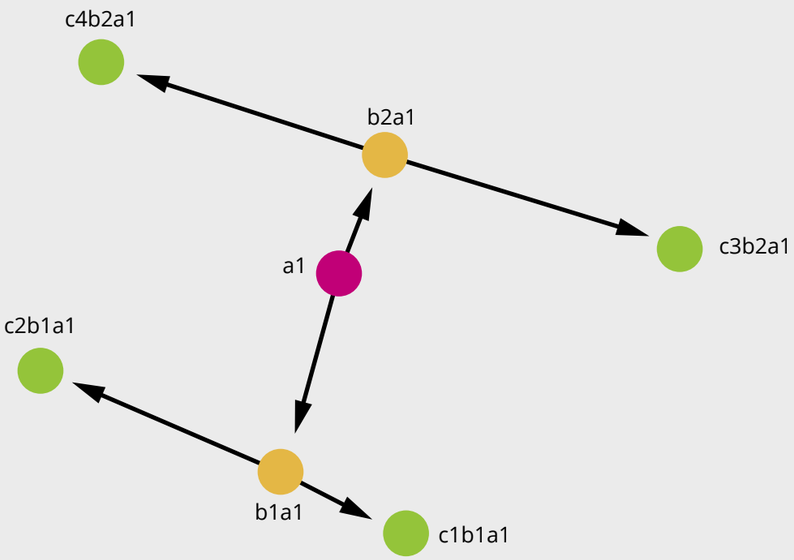
```
import networkx as nx
import matplotlib.pyplot as plt
from networkx.drawing.nx_pydot import graphviz_layout
import operator
```

```
dir(nx.Graph()) # metody networkx
[….
 'add_edge',
 'add_edges_from',
 'add_node',
 'add_nodes_from',
 'add_weighted_edges_from',
 'adj',
 'adjacency',
 'adjlist_inner_dict_factory',
 'adjlist_outer_dict_factory',
 'clear',
 'clear_edges',
 'copy',
 'degree',
 'edge_attr_dict_factory',
 'edge_subgraph',
 'edges',
 'get_edge_data',
 'graph',
```

```
 'graph_attr_dict_factory',
 'has_edge',
 'has_node',
 'is_directed',
 'is_multigraph',
 'name',
 'nbunch_iter',
 'neighbors',
 'node_attr_dict_factory',
 'node_dict_factory',
 'nodes',
 'number_of_edges',
 'number_of_nodes',
 'order',
 'remove_edge',
 'remove_edges_from',
 'remove_node',
 'remove_nodes_from',
 'size',
 'subgraph',
 'to_directed',
 'to_directed_class',
 'to_undirected',
 'to_undirected_class',
 'update']


můj_graf = nx.DiGraph() # vytváříme naši instanci grafu

můj_graf.add_nodes_from(['a1','b1a1','b2a1','c1b1a1','c2b1a1','c3b2a1','c4b2a1']) #
přidáme nódy

print(f"Tento můj graf má nyní {můj_graf.number_of_nodes()} nódů.")
Tento můj graf má nyní 7 nódů.

# přidáme strany grafu a váhy

můj_graf.add_edge("a1","b1a1",weight=3.0)
můj_graf.add_edge("a1","b2a1", weight=10.0)
můj_graf.add_edge("b1a1","c1b1a1", weight=100.0)
```

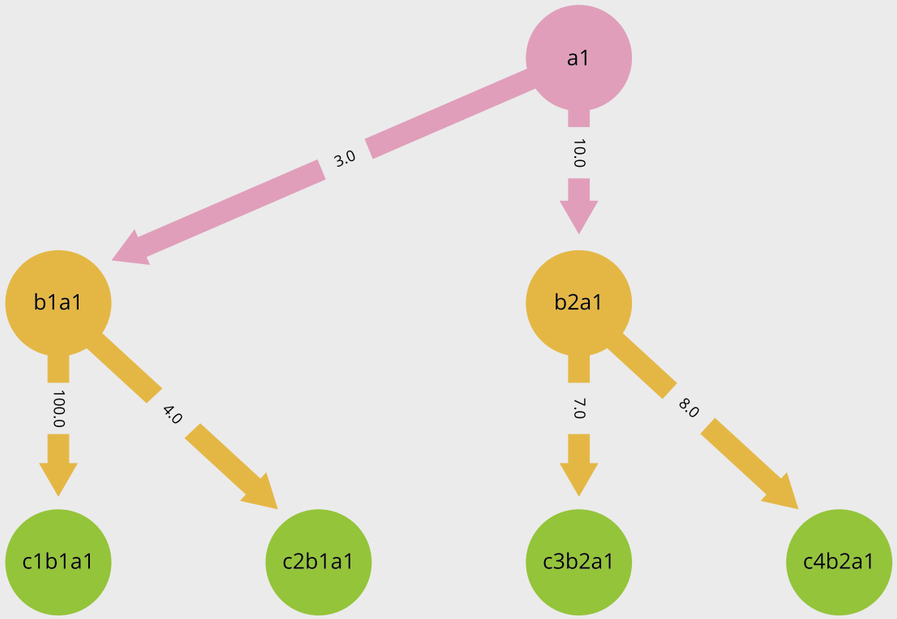


```
můj_graf.add_edge("b1a1","c2b1a1", weight=4.0)
můj_graf.add_edge("b2a1","c3b2a1", weight=7.0)
můj_graf.add_edge("b2a1","c4b2a1", weight=8.0)

color_list = ["violet", "darkorange", "darkorange",
 "limegreen", "limegreen", "limegreen","limegreen"]

nx.draw_networkx(můj_graf, node_color=color_list, with_labels=True) # nakreslíme si
náš graf
```



A trochu hezčí graf našeho stromu

```
edge_color_list = ["violet", "violet",
 "darkorange", "darkorange", "darkorange","darkorange"]

pos = graphviz_layout(můj_graf, prog="dot")

nx.draw(můj_graf, pos, node_color=color_list, edge_color = edge_color_list, width=5,
node_size=2000, with_labels=True)

labels = nx.get_edge_attributes(můj_graf,'weight')
nx.draw_networkx_edge_labels(můj_graf, pos, edge_labels=labels)
plt.show()
```

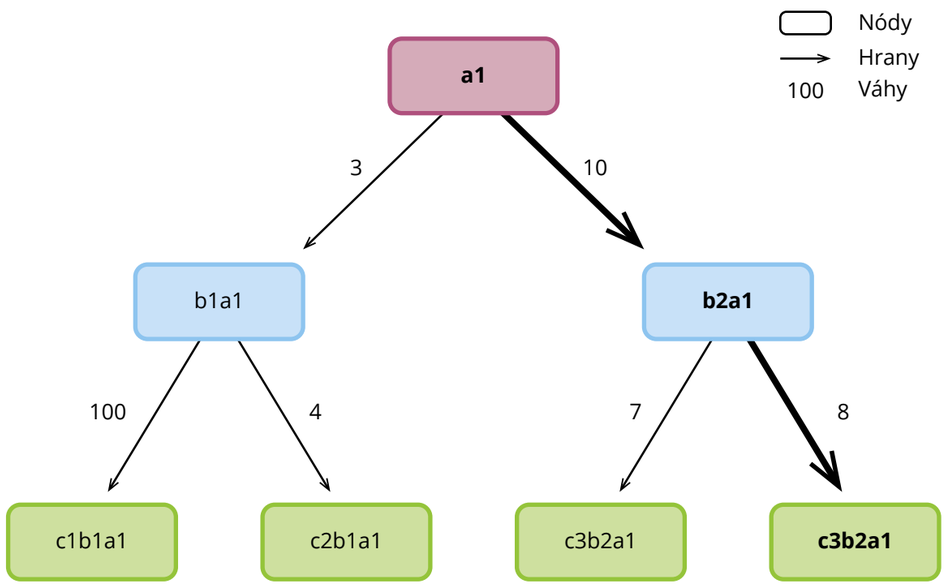



```
nodes = list(můj_graf.nodes)
nodes
['a1', 'b1a1', 'b2a1', 'c1b1a1', 'c2b1a1', 'c3b2a1', 'c4b2a1']
edges=list(můj_graf.edges)
edges
[('a1', 'b1a1'),
 ('a1', 'b2a1'),
 ('b1a1', 'c1b1a1'),
 ('b1a1', 'c2b1a1'),
 ('b2a1', 'c3b2a1'),
 ('b2a1', 'c4b2a1')]
můj_graf.edges[('a1', 'b1a1')]
{'weight': 3.0}
list(můj_graf.neighbors('c4b2a1'))
[]

můj_graf['a1']['b1a1']['weight']
3.0
```


Hladový algoritmus ve stromu

Příklad řešení hladového algoritmu ve stromu.

Nejprve sestavíme jednoduchý strom, co je na obrázku.



Strom s váhami hladový algoritmus

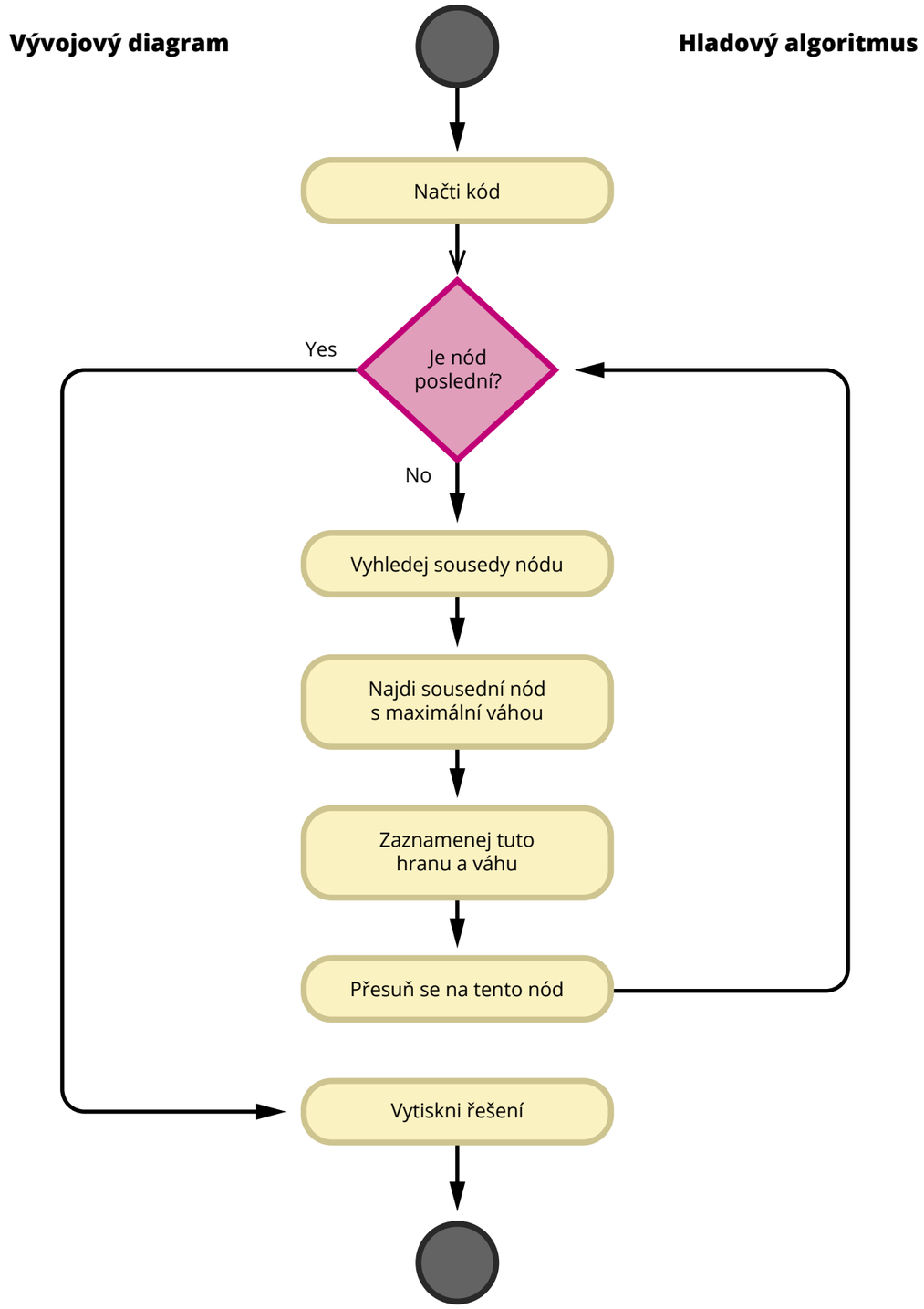
Formulace problému

Hladový algoritmus si v každé části cesty vybírá nejziskovější trasu. To sice ve finále nemusí být nejvýhodnější, ale takto funguje.

Analýza úlohy

Začínáme u kořene stromu a pokračujeme k nejbližšímu sousedovi s nejlepší váhou. Končíme, když nód nemá žádné další sousedy.

Vytvoření algoritmu úlohy - vývojový diagram



Sestavení programu a jeho odladění

```
import networkx as nx
import operator



můj_graf = nx.DiGraph()
můj_graf.add_nodes_from(['a1','b1a1','b2a1','c1b1a1','c2b1a1','c3b2a1','c4b2a1'])

můj_graf.add_edge("a1","b1a1",weight=3.0)
můj_graf.add_edge("a1","b2a1", weight=10.0)
můj_graf.add_edge("b1a1","c1b1a1", weight=100.0)
můj_graf.add_edge("b1a1","c2b1a1", weight=4.0)
můj_graf.add_edge("b2a1","c3b2a1", weight=7.0)
můj_graf.add_edge("b2a1","c4b2a1", weight=8.0)

node=list(můj_graf.nodes)[0]
while True:
 neighbors=list(můj_graf.neighbors(node))
 if len(neighbors) == 0:
 break
 max_weight=0
 branch={}
 for neighbor in neighbors:
 branch[(node, neighbor)] = můj_graf[node][neighbor]['weight']
 print(max(branch.items(), key=operator.itemgetter(1)))
 node = max(branch.items(), key=operator.itemgetter(1))[0][1]



(('a1', 'b2a1'), 10.0)
(('b2a1', 'c4b2a1'), 8.0)
```

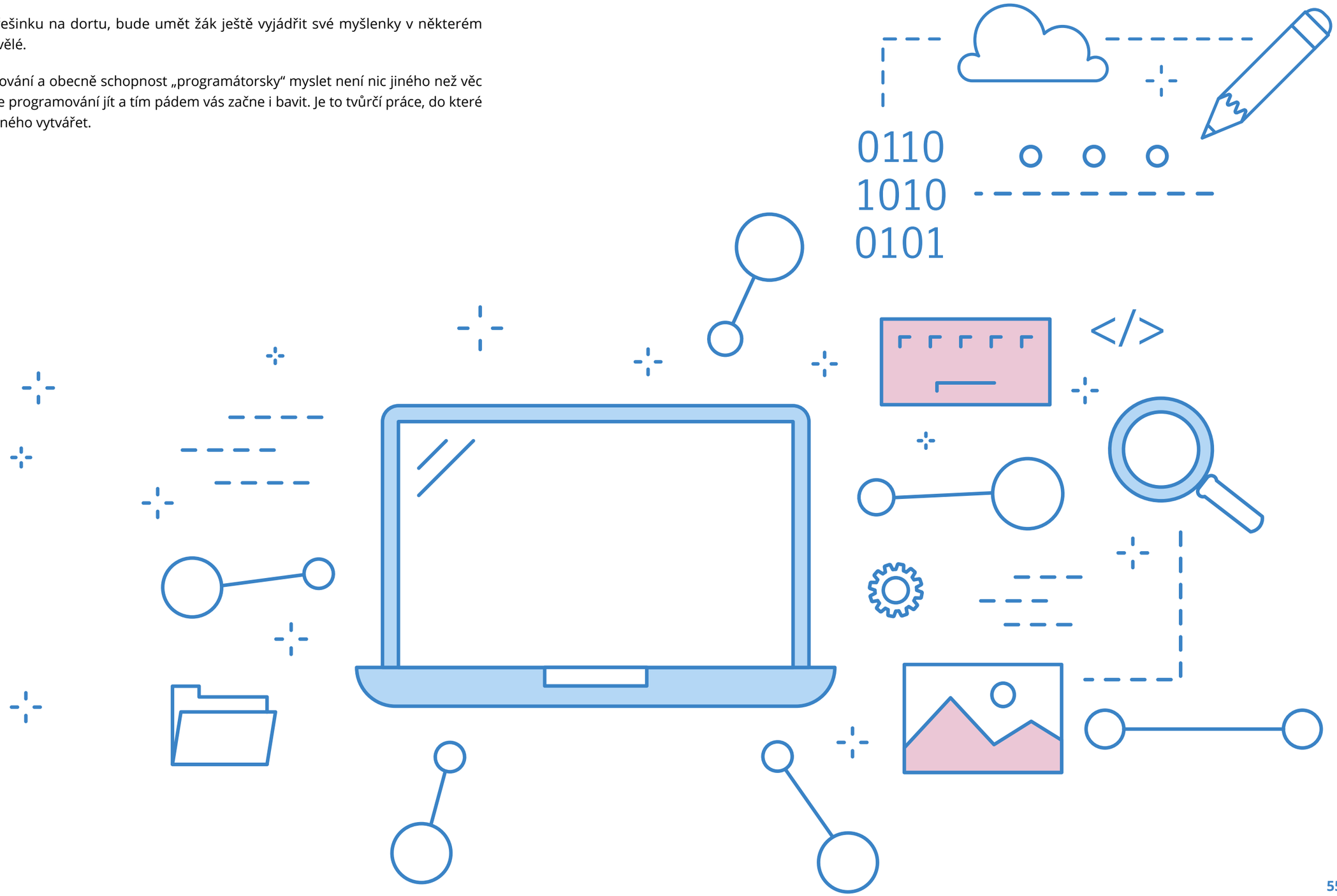
Co jste se naučili

Schopnost „programátorsky“ myslet patří dnes k důležitým dovednostem. Pokud tento kurz přispěje k tomu, aby učitelé i žáci uměli formulovat problém, analyzovat ho, zakreslit jednotlivé kroky řešení a rozhodování v programu a předvídání základních druhů chyb, které by při běhu mohly nastat, pak jsem spokojená.

Pokud k tomu, jako pověstnou třešinku na dortu, bude umět žák ještě vyjádřit své myšlenky v některém programovacím jazyce, bude to skvělé.

Stejně jako hra na piano, programování a obecně schopnost „programátorsky“ myslet není nic jiného než věc tréninku. Po určité době vám začne programování jít a tím pádem vás začne i bavit. Je to tvůrčí práce, do které můžete dát něco svého a něco cenného vytvářet.

Držím palce.



IMPULS
PRO KARIÉRU
A PRAXI