

Základy Pythonu, typy, jednoduché algoritmy

IB111 ZÁKLADY PROGRAMOVÁNÍ

Tomáš Vojnar (na základě slajdů Nikoly Beneše)

26. září 2025

Sekvenční řazení příkazů

- Pod sebe, **jeden příkaz na řádek**.
- Více příkazů na jednom řádku můžeme oddělit **středníkem** ;.
 - Nedoporučeno používat, nepřehledné, zhoršuje čitelnost.
- Příliš dlouhý řádek můžeme **rozdělit** na více řádků:
 - pokračování řádku naznačíme znakem \ na konci řádku (není třeba uvnitř závorek).

Zanořování podmínek/cyklů

- Tělo **odsazujeme**, doporučeny 4 mezery.
- Můžeme **zanořovat**:
 - podmínka uvnitř cyklu (a naopak),
 - cyklus uvnitř cyklu,
 - podmínka v podmínce.
 - Libovolný počet zanoření (ale zhoršuje to čitelnost).

- Minule jsme viděli `for i in range(10)`.
- `for` v Pythonu je obecnější:
 - `range(...)` je jeden z druhů objektů, kterými se dá procházet.
 - Dalšími jsou seznamy, množiny, ... (uvidíme dále).

`range` (česky snad sekvence, posloupnost, rozsah)

- Základní verze `range(stop)`:
 - postupně procházíme čísla od `0` do `stop - 1` včetně.
- `range(start, stop)`:
 - procházíme čísla od `start` do `stop - 1` včetně.
- `range(start, stop, step)`:
 - procházíme čísla od `start`, velikost jednoho kroku je `step`, skončíme na posledním čísle *před* `stop`.
- Jak můžeme počítat pozpátku?

- Minule jsme viděli `for i in range(10)`.
- `for` v Pythonu je obecnější:
 - `range(...)` je jeden z druhů objektů, kterými se dá procházet.
 - Dalšími jsou seznamy, množiny, ... (uvidíme dále).

`range` (česky snad sekvence, posloupnost, rozsah)

- Základní verze `range(stop)`:
 - postupně procházíme čísla od `0` do `stop - 1` včetně.
- `range(start, stop)`:
 - procházíme čísla od `start` do `stop - 1` včetně.
- `range(start, stop, step)`:
 - procházíme čísla od `start`, velikost jednoho kroku je `step`, skončíme na posledním čísle *před* `stop`.
- Jak můžeme počítat pozpátku? Použijeme záporný krok.

```
total = 0
for i in range(10, 100, 7):
    total += i
```

*Jak napsat cyklus **for** pomocí cyklu **while**?*

```
total = 0
for i in range(10, 100, 7):
    total += i
```

*Jak napsat cyklus **for** pomocí cyklu **while**?*

```
total = 0
i = 10
while i < 100:
    total += i
    i += 7
```

K zamyšlení: Mezi těmito dvěma cykly je drobný rozdíl, v čem?

```
total = 0
for i in range(10, 100, 7):
    total += i
```

Jak napsat cyklus `for` pomocí cyklu `while`?

```
total = 0
i = 10
while i < 100:
    total += i
    i += 7
```

K zamyšlení: Mezi těmito dvěma cykly je drobný rozdíl, v čem?

- Jaká je hodnota proměnné `i` po skončení cyklů?
 - Tento rozdíl v některých jazycích nenajdete.
 - Je lepší se na hodnotu `i` po skončení cyklu `for` nespolehat.

Ternární operátor `if ... else`

- Už známe: **příkaz** `if`.
- Python má také **operátor** `if ... else`.
 - Podobné existují v jiných jazycích, např. `?:` v C.
 - V Haskellu `if ... then ... else`.

výraz1 `if` podmínka `else` výraz2

- Nejprve se vyhodnotí podmínka.
- Pokud je pravdivá, vyhodnotí se výraz1.
- V opačném případě se vyhodnotí výraz2.
- Část `else ...` je v tomto případě povinná.

Příkaz vs. výraz

- Pojem výraz: viz předchozí přednáška.
- Výraz (ale ne příkaz) můžeme použít uvnitř jiného výrazu.
- výraz1 a výraz2 jsou výrazy, ne příkazy!

- Programy dělíme na podprogramy.
- Ty můžeme dále dělit na (menší) podprogramy.
- Důvody už jsme si řekli minule.

- Programy dělíme na podprogramy.
- Ty můžeme dále dělit na (menší) podprogramy.
- Důvody už jsme si řekli minule.
- **Podprogram**: úsek kódu, který má navíc název, parametry a návratovou hodnotu.
- V Pythonu realizujeme pomocí **funkcí**:

```
def ppgm(x_1, ..., x_n):  
    příkaz_1  
    # ...  
    příkaz_m
```

- ppgm je **jméno**.
- x_i jsou jména **formálních parametrů**,
- příkaz_1 ... příkaz_m je **tělo** podprogramu.

- **Volání podprogramu** – výraz `ppgm(e_1, ..., e_m)`:
 - `e_i` – výrazy: vyhodnotí se na **skutečné parametry** (argumenty),
 - jména formálních parametrů se **sváží** se skutečnými parametry.
- *Příkaz* **return** či **return** `expr`.
 - Ukončí běh podprogramu, vrátí **None** či hodnotu výrazu `expr`.
 - Návratová hodnota se použije místo (pod)výrazu volání podprogramu.

- **Volání podprogramu** – výraz `ppgm(e_1, ..., e_m)`:
 - `e_i` – výrazy: vyhodnotí se na **skutečné parametry** (argumenty),
 - jména formálních parametrů se **sváží** se skutečnými parametry.
- *Příkaz* **return** či **return** `expr`.
 - Ukončí běh podprogramu, vrátí **None** či hodnotu výrazu `expr`.
 - Návrátová hodnota se použije místo (pod)výrazu volání podprogramu.
- **Vnořené volání podprogramů**
 - Podprogramy mohou volat další podprogramy.
 - Po dokončení vnořného podprogramu se běh vrátí do místa, odkud se podprogram zavolal.
 - Podprogram může dokonce volat sám sebe:
 - tzv. *rekurze* (k té se vrátíme později).

```
def series_sum(num):  
    total = 0  
    for i in range(1, num + 1):  
        total += i  
    return total
```

what is the value of series_sum(42)?

- Lepší řešení?

```
def series_sum(num):  
    total = 0  
    for i in range(1, num + 1):  
        total += i  
    return total
```

what is the value of series_sum(42)?

- Lepší řešení?

```
return num * (num + 1) // 2
```

- Jejich jméno je viditelné lokálně pouze v daném podprogramu:
 - buď se jedná o **formální parametr**,
 - nebo **vznikají lokálním přiřazením**.
- **Každá invokace funkce má své vlastní lokální proměnné.**
 - Stejné jméno, ale jiný kontext!
 - Platí i pro rekurzivní invokace.

use pythontutor.com or a debugger

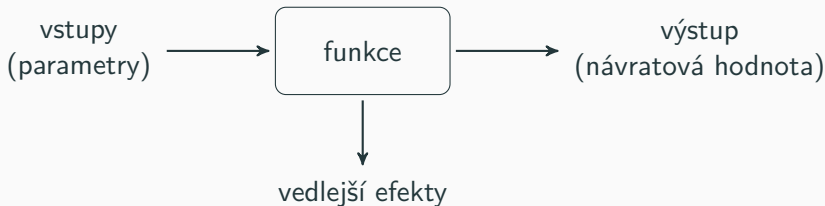
```
def inner(n):  
    x = 2 * n  
    return x + 11
```

```
def outer(n):  
    x = 3 * n  
    y = inner(n + 7)  
    return x + y
```

```
x = 17  
n = 33  
result = outer(9)
```


Vedlejší efekty podprogramů

- Mimo vrácení návratové hodnoty podprogram může mít další **vně pozorovatelné efekty**.
- Chápání se může lišit, ale typicky jde o změnu:
 - hodnoty objektu dostupného v programu i **vně daného podprogramu** (pokud to typ objektu povoluje),
 - stavu výpočetního systému **vně daného běžícího programu**.
 - Změna obsahu souborů, jejich atributů, přístupových časů, ...
 - Obvykle ale nepočítáme spotřebu paměti, energie apod.



Čistá funkce – pokud při jejím provádění nenastane chyba mimo funkci samotnou (výpadek napájení, nedostatek paměti, ...), pak:

- pro stejné vstupy buď vždy skončí, nebo vždy neskončí;
- pokud skončí, pak **skončí se stejnými hodnotami**;
- nemá mimo svého provedení a vrácení výsledku **žádný vně pozorovatelný efekt**, a to ani dočasně.

Čistá funkce – pokud při jejím provádění nenastane chyba mimo funkci samotnou (výpadek napájení, nedostatek paměti, ...), pak:

- pro stejné vstupy buď vždy skončí, nebo vždy neskončí;
- pokud skončí, pak **skončí se stejnými hodnotami**;
- nemá mimo svého provedení a vrácení výsledku **žádný vně pozorovatelný efekt**, a to ani dočasně.

Nečistota funkce může být způsobena i podprogramem z ní volaným!

- Ale pozor: neplatí to vždy!

Čistá funkce – pokud při jejím provádění nenastane chyba mimo funkci samotnou (výpadek napájení, nedostatek paměti, ...), pak:

- pro stejné vstupy buď vždy skončí, nebo vždy neskončí;
- pokud skončí, pak **skončí se stejnými hodnotami**;
- nemá mimo svého provedení a vrácení výsledku **žádný vně pozorovatelný efekt**, a to ani dočasně.

Nečistota funkce může být způsobena i podprogramem z ní volaným!

- Ale pozor: neplatí to vždy!

Predikát: čistá funkce, která vrátí **True** nebo **False**

Procedura: podprogram, jehož hlavní účel je nějaký vedlejší efekt.

Nejprve si ujasnit **specifikaci** (vstupně/výstupní chování):

- Jaké potřebuje funkce vstupy?
- Co bude výstupem funkce?

Funkce by měly být **krátké**.

- „Jedna myšlenka.“
- Funkce by se měla vejít na jednu obrazovku.
- Jen několik úrovní zanoření.

Co když je funkce příliš dlouhá?

Nejprve si ujasnit **specifikaci** (vstupně/výstupní chování):

- Jaké potřebuje funkce vstupy?
- Co bude výstupem funkce?

Funkce by měly být **krátké**.

- „Jedna myšlenka.“
- Funkce by se měla vejít na jednu obrazovku.
- Jen několik úrovní zanoření.

Co když je funkce příliš dlouhá?

- Najít v ní menší logické celky.
- Rozdělit na menší funkce.

Co je hodnotou výrazu `3 + "ježek"` v Pythonu?

Co je hodnotou výrazu `3 + "ježek"` v Pythonu?

`TypeError: unsupported operand type(s) for +: 'int' and 'str'`

- Hodnoty uložené v objektech mají své **typy**.
- Většinu operací v Pythonu je možno provádět jen se správnými (kompatibilními) typy hodnot.

Co je hodnotou výrazu `3 + "ježek"` v Pythonu?

`TypeError: unsupported operand type(s) for +: 'int' and 'str'`

- Hodnoty uložené v objektech mají své **typy**.
- Většinu operací v Pythonu je možno provádět jen se správnými (kompatibilními) typy hodnot.
- Velká část jazyků vyžaduje **typové deklarace** proměnných:
 - proměnná smí obsahovat hodnoty jen zadaného typu,
 - typová kontrola se provádí před spuštěním programu.
- Proměnné v Pythonu mohou mít hodnoty libovolných typů.
 - Python je **silně typovaný dynamický jazyk**.
 - Typová kontrola se provádí za běhu!
- Python (od verze 3.6) má nepovinné **typové anotace**.
 - Umožňují typovou kontrolu před spuštěním programu.
 - Mohou značně zjednodušit orientaci v kódu.

Základní typy hodnot v Pythonu

- Celá čísla – `int`:
 - příklady: `1`, `-7`, `36893488147419103232`,
 - od Pythonu 3 **libovolně velká**,
 - v jiných jazycích často omezeně velká (32 bitů, 64 bitů).
 - Dvojkově: `0b101010`, osmičkově: `0o52`, šestnáctkově: `0x2a`.
- Čísla s plovoucí řádovou čárkou – `float`:
 - příklady: `1.0`, `3.14`, `3.68e+19` (tj. $3,68 \cdot 10^{19}$),
 - **omezená přesnost** (IEEE floating point).
- Řetězce – `str`:
 - příklady: `"Apple"`, `'Banana'`, `"Cherry"`
 - víceřádkové uvnitř `""" ... """`
- Pravdivostní hodnoty – `bool`:
 - dvě hodnoty: `True`, `False`.
- Seznamy, ntice, slovníky, ... (uvidíme později)

None

- Vraceno funkcí, pokud se dojde na její konec bez `return`.
- Vraceno prázdným `return`.
- Explicitní označení „žádné hodnoty“.
- V případě, že funkce může vracet i jiné hodnoty, je čitelnější používat `return None`.

Test na *None* (dle PEP8)

- `x is None` nebo `x is not None`.
- Jinde `is nepoužívejte` – není to totéž, co `==`.
 - (Vrátíme se k tomu později.)

Implicitní (dějí se automaticky)

- Celé číslo \rightarrow číslo v plovoucí řádové čárce.¹
- Pravdivostní hodnota \rightarrow číslo.

Explicitní (je třeba si o ně říct)

- `float(num)` – z celého čísla na typ `float`.
- `round(fnum)` – „klasické“ zaokrouhlení (k bližšímu číslu);
 - pokud je `fnum` přesně mezi, výsledek bude sudý.
- V knihovně `math`:
 - `floor(fnum)` – dolní celá část (zaokrouhlení dolů),
 - `ceil(fnum)` – horní celá část (zaokrouhlení nahoru),
 - `trunc(fnum)` – odříznutí desetinné části (zaokrouhlení k nule).
- Další uvidíme později.

¹Ale pozor na `/`, v `IB111` nedovolujeme celá čísla na obou stranách.

Obecné

- Pište tak, aby byl jasný záměr:
 - smysluplné pojmenování proměnných, funkcí apod.;
 - v případě komplikovanějšího kódu dokumentace (komentáře)
 - (ideálně ovšem psát samodokumentující se kód).
- **Neopakujte se**, nepište „copy&paste kód“.
- Větší kód čleňte do rozumných menších celků:
 - pomocné funkce, (objekty, moduly, ...).

PEP8 – oficiální styl pro Python

- Část pravidel zkontroluje IDE nebo edulint.
- Konvence pro jména:
 - proměnné, funkce: `lower_case_with_underscores`,
 - konstanty: `UPPER_CASE_WITH_UNDERSCORES`.



Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live. (J.F. Woods)

DRY

- Don't Repeat Yourself

WET

- Write Everything Twice
- We Enjoy Typing
- Waste Everyone's Time

Chceme napsat predikát `is_leap`, který nám řekne, zda je zadaný rok přestupný v gregoriánském kalendáři.

Pravidla:

- Rok je přestupný, pokud je dělitelný číslem 4.
- Výjimka: Rok není přestupný, pokud je dělitelný 100.
- Výjimka z výjimky: Rok je přestupný, pokud je dělitelný 400.


```
def is_leap(year):  
    if year % 4 == 0:  
        if year % 100 == 0:  
            if year % 400 == 0:  
                return True  
            else:  
                return False  
        else:  
            return True  
    else:  
        return False
```

```
def is_leap(year):  
    if year % 4 == 0:  
        if year % 100 == 0:  
            if year % 400 == 0:  
                return True  
            else:  
                return False  
        else:  
            return True  
    else:  
        return False
```

A vám se ten kód jako líbí?

```
def is_leap(year):  
    return year % 4 == 0 and \  
        (year % 100 != 0 or year % 400 == 0)
```

- *Vstup*: přirozené číslo n .
- *Výstup*: ciferný součet čísla n (v desítkové soustavě).
- Příklady konkrétního vstupu a výstupu:
 - $0 \rightarrow 0$
 - $7 \rightarrow 7$
 - $17 \rightarrow 8$
 - $42 \rightarrow 6$
 - $999 \rightarrow 27$
 - $72525 \rightarrow 21$
- Jak na to?

- *Vstup*: přirozené číslo n .
- *Výstup*: ciferný součet čísla n (v desítkové soustavě).
- Příklady konkrétního vstupu a výstupu:
 - $0 \rightarrow 0$
 - $7 \rightarrow 7$
 - $17 \rightarrow 8$
 - $42 \rightarrow 6$
 - $999 \rightarrow 27$
 - $72525 \rightarrow 21$
- Jak na to?
 - Potřebujeme umět „odebrat“ jednu číslici z čísla.

- Poslední číslice z čísla: zbytek po dělení deseti.
- Odebrání poslední číslice: celočíselné dělení deseti.
- Opakovat tak dlouho, dokud číslo není 0.

```
def digit_sum(n):  
    result = 0  
    while n > 0:  
        # MAGIC happens...  
        n = n // 10  
    return result
```

```
if n % 10 == 1:
    result = result + 1
elif n % 10 == 2:
    result = result + 2
elif n % 10 == 3:
    result = result + 3
elif n % 10 == 4:
    result = result + 4
elif n % 10 == 5:
    result = result + 5
elif n % 10 == 6:
    result = result + 6
elif n % 10 == 7:
    result = result + 7
elif n % 10 == 8:
    result = result + 8
elif n % 10 == 9:
    result = result + 9
```

```
def digit_sum(n):  
    result = 0  
    while n > 0:  
        result += n % 10  
        n //= 10  
    return result
```



```
def digit_sum(n):  
    result = 0  
    while n > 0:  
        result += n % 10  
        n //= 10  
    return result
```

*Co kdybychom chtěli ciferný součet v jiné soustavě?
(Dvojkové, osmičkové, šestnáctkové, dvanáctkové, šedesátkové, ...)*

Jak spočítáte druhou odmocninu kladného (reálného) čísla?

- Bez použití knihovny `math` a funkce `sqrt`.
- Co kdyby nám stačilo jen *přibližné* řešení?

Jak spočítáte druhou odmocninu kladného (reálného) čísla?

- Bez použití knihovny `math` a funkce `sqrt`.
- Co kdyby nám stačilo jen *přibližné* řešení?
- *Vstup*: kladné („reálné“) číslo x .
- *Výstup*: přibližná hodnota \sqrt{x} .
- Co to znamená *přibližná*?
- Jak na to?

Jak spočítáte druhou odmocninu kladného (reálného) čísla?

- Bez použití knihovny `math` a funkce `sqrt`.
- Co kdyby nám stačilo jen *přibližné* řešení?
- *Vstup*: kladné („reálné“) číslo x .
- *Výstup*: přibližná hodnota \sqrt{x} .
- Co to znamená *přibližná*?
- Jak na to?
 - Mnoho různých metod, ukážeme si jednu z nich (zdaleka ne tu nejefektivnější).

Přesná matematika – pro $x \neq 0$:

$$\left(\left(1 + \frac{1}{x} \right) - 1 \right) \cdot x = 1$$

Nepřesné počítače:

```
def weird(x):
    return ((1 + 1.0 / x) - 1) * x
```

```
# weird(2 ** 50) == 1.0
# weird(2 ** 100) == 0.0
# weird(3) == 0.9999999999999999
```

Pro které nejmenší kladné číslo x platí $1.0 / x + 1.0 == 1.0$?

Pro které nejmenší kladné číslo x platí $x / 1.0 + 1.0 == x$?

Půlení intervalu

- Chceme vypočítat $\sqrt{2}$.



Půlení intervalu

- Chceme vypočítat $\sqrt{2}$.
- Řešení musí být v intervalu $\langle 0, 2 \rangle$.



Půlení intervalu

- Chceme vypočítat $\sqrt{2}$.
- Řešení musí být v intervalu $\langle 0, 2 \rangle$.
- Zvolíme střed intervalu, tj. 1.



Půlení intervalu

- Chceme vypočítat $\sqrt{2}$.
- Řešení musí být v intervalu $\langle 0, 2 \rangle$.
- Zvolíme střed intervalu, tj. 1.
- $1^2 = 1$, což je méně než 2.



Půlení intervalu

- Chceme vypočítat $\sqrt{2}$.
- Řešení musí být v intervalu $\langle 0, 2 \rangle$.
- Zvolíme střed intervalu, tj. 1.
- $1^2 = 1$, což je méně než 2.
- Řešení tedy musí být v intervalu $\langle 1, 2 \rangle$.



Půlení intervalu

- Chceme vypočítat $\sqrt{2}$.
- Řešení musí být v intervalu $\langle 0, 2 \rangle$.
- Zvolíme střed intervalu, tj. 1.
- $1^2 = 1$, což je méně než 2.
- Řešení tedy musí být v intervalu $\langle 1, 2 \rangle$.
- Zvolíme střed intervalu, tj. 1.5.



Půlení intervalu

- Chceme vypočítat $\sqrt{2}$.
- Řešení musí být v intervalu $\langle 0, 2 \rangle$.
- Zvolíme střed intervalu, tj. 1.
- $1^2 = 1$, což je méně než 2.
- Řešení tedy musí být v intervalu $\langle 1, 2 \rangle$.
- Zvolíme střed intervalu, tj. 1.5.
- $1.5^2 = 2.25$, což je více než 2.



Půlení intervalu

- Chceme vypočítat $\sqrt{2}$.
- Řešení musí být v intervalu $\langle 0, 2 \rangle$.
- Zvolíme střed intervalu, tj. 1.
- $1^2 = 1$, což je méně než 2.
- Řešení tedy musí být v intervalu $\langle 1, 2 \rangle$.
- Zvolíme střed intervalu, tj. 1.5.
- $1.5^2 = 2.25$, což je více než 2.
- Řešení tedy musí být v intervalu $\langle 1, 1.5 \rangle$.



Půlení intervalu

- Chceme vypočítat $\sqrt{2}$.
- Řešení musí být v intervalu $\langle 0, 2 \rangle$.
- Zvolíme střed intervalu, tj. 1.
- $1^2 = 1$, což je méně než 2.
- Řešení tedy musí být v intervalu $\langle 1, 2 \rangle$.
- Zvolíme střed intervalu, tj. 1.5.
- $1.5^2 = 2.25$, což je více než 2.
- Řešení tedy musí být v intervalu $\langle 1, 1.5 \rangle$.
- Atd.



Půlení intervalu

- Chceme vypočítat $\sqrt{2}$.
- Řešení musí být v intervalu $\langle 0, 2 \rangle$.
- Zvolíme střed intervalu, tj. 1.
- $1^2 = 1$, což je méně než 2.
- Řešení tedy musí být v intervalu $\langle 1, 2 \rangle$.
- Zvolíme střed intervalu, tj. 1.5.
- $1.5^2 = 2.25$, což je více než 2.
- Řešení tedy musí být v intervalu $\langle 1, 1.5 \rangle$.
- Atd.



Půlení intervalu

- Chceme vypočítat $\sqrt{2}$.
- Řešení musí být v intervalu $\langle 0, 2 \rangle$.
- Zvolíme střed intervalu, tj. 1.
- $1^2 = 1$, což je méně než 2.
- Řešení tedy musí být v intervalu $\langle 1, 2 \rangle$.
- Zvolíme střed intervalu, tj. 1.5.
- $1.5^2 = 2.25$, což je více než 2.
- Řešení tedy musí být v intervalu $\langle 1, 1.5 \rangle$.
- Atd.



Půlení intervalu

- Chceme vypočítat $\sqrt{2}$.
- Řešení musí být v intervalu $\langle 0, 2 \rangle$.
- Zvolíme střed intervalu, tj. 1.
- $1^2 = 1$, což je méně než 2.
- Řešení tedy musí být v intervalu $\langle 1, 2 \rangle$.
- Zvolíme střed intervalu, tj. 1.5.
- $1.5^2 = 2.25$, což je více než 2.
- Řešení tedy musí být v intervalu $\langle 1, 1.5 \rangle$.
- Atd.



Půlení intervalu

- Chceme vypočítat $\sqrt{2}$.
- Řešení musí být v intervalu $\langle 0, 2 \rangle$.
- Zvolíme střed intervalu, tj. 1.
- $1^2 = 1$, což je méně než 2.
- Řešení tedy musí být v intervalu $\langle 1, 2 \rangle$.
- Zvolíme střed intervalu, tj. 1.5.
- $1.5^2 = 2.25$, což je více než 2.
- Řešení tedy musí být v intervalu $\langle 1, 1.5 \rangle$.
- Atd.



Půlení intervalu

- Chceme vypočítat $\sqrt{2}$.
- Řešení musí být v intervalu $\langle 0, 2 \rangle$.
- Zvolíme střed intervalu, tj. 1.
- $1^2 = 1$, což je méně než 2.
- Řešení tedy musí být v intervalu $\langle 1, 2 \rangle$.
- Zvolíme střed intervalu, tj. 1.5.
- $1.5^2 = 2.25$, což je více než 2.
- Řešení tedy musí být v intervalu $\langle 1, 1.5 \rangle$.
- Atd.



Půlení intervalu

- Chceme vypočítat $\sqrt{2}$.
- Řešení musí být v intervalu $\langle 0, 2 \rangle$.
- Zvolíme střed intervalu, tj. 1.
- $1^2 = 1$, což je méně než 2.
- Řešení tedy musí být v intervalu $\langle 1, 2 \rangle$.
- Zvolíme střed intervalu, tj. 1.5.
- $1.5^2 = 2.25$, což je více než 2.
- Řešení tedy musí být v intervalu $\langle 1, 1.5 \rangle$.
- Atd.



Půlení intervalu

- Chceme vypočítat $\sqrt{2}$.
- Řešení musí být v intervalu $\langle 0, 2 \rangle$.
- Zvolíme střed intervalu, tj. 1.
- $1^2 = 1$, což je méně než 2.
- Řešení tedy musí být v intervalu $\langle 1, 2 \rangle$.
- Zvolíme střed intervalu, tj. 1.5.
- $1.5^2 = 2.25$, což je více než 2.
- Řešení tedy musí být v intervalu $\langle 1, 1.5 \rangle$.
- Atd.



- Kdy skončíme?

Půlení intervalu

- Chceme vypočítat $\sqrt{2}$.
- Řešení musí být v intervalu $\langle 0, 2 \rangle$.
- Zvolíme střed intervalu, tj. 1.
- $1^2 = 1$, což je méně než 2.
- Řešení tedy musí být v intervalu $\langle 1, 2 \rangle$.
- Zvolíme střed intervalu, tj. 1.5.
- $1.5^2 = 2.25$, což je více než 2.
- Řešení tedy musí být v intervalu $\langle 1, 1.5 \rangle$.
- Atd.



- Kdy skončíme?
 - Když bude střed na druhou dostatečně blízko 2.

```
def square_root(x, precision):  
    lower = 0.0  
    upper = x  
    middle = (lower + upper) / 2  
    while abs(middle ** 2 - x) > precision:  
        if middle ** 2 > x:  
            upper = middle  
        elif middle ** 2 < x:  
            lower = middle  
        middle = (lower + upper) / 2  
    return middle
```

- Je tento program korektní?

```
def square_root(x, precision):  
    lower = 0.0  
    upper = x  
    middle = (lower + upper) / 2  
    while abs(middle ** 2 - x) > precision:  
        if middle ** 2 > x:  
            upper = middle  
        elif middle ** 2 < x:  
            lower = middle  
        middle = (lower + upper) / 2  
    return middle
```

- Je tento program korektní?
 - Zkuste spočítat druhou odmocninu čísla 0.5.
 - Co se stalo a proč?
 - Jak to opravit?

9/9

0800 Antan started
 1000 " stopped - antan ✓
 1300 (032) MP-MC ~~1.58267000~~ 2.130476415
 (033) PRO 2 2.130476415
 convd 2.130676415

{ 1.2700 9.037847025
 9.037846795 convd
 4.615925059(-2)

Relays 6-2 in 033 failed special speed test
 in relay " 11.000 test.

Relay
 3145
 Relay 3376

1100 Relays changed
 Started Cosine Tape (Sine check)
 1525 Started Multi-Adder Test.

1545



Relay #70 Panel F
 (moth) in relay.

First actual case of bug being found.
 1630 Antan started.
 1700 closed down.

„Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.“ (Brian W. Kernighan)

Debugger

- Nástroj, pomocí něhož je možné postupně provádět jednotlivé kroky programu, sledovat hodnoty proměnných apod.
- Typicky součástí vývojového prostředí.
- Příkazový řádek: `python -m pdb jméno_souboru.py`.

Online vizualizátor: <http://pythontutor.com/>

- Nastavení řádků, kde se má výpočet zastavit – **breakpoints**:
 - kliknout na číslo řádku, objeví se červená tečka.
- Spuštění programu s debuggerem:
 - místo Run použít **Debug**,
 - nebo v menu Run → Debug current script (dvě varianty: *nicer*/*faster*, liší se detailem Step Into).
- Hodnoty proměnných:
 - Globální (přesněji modulové): menu View → **Variables**.
 - Lokální: v okně s funkcí (**Local variables**).
- Krokování:
 - **Step Over** – krok na další řádek,
 - **Step Into** – vstoupí dovnitř volání funkce (ve variantě *nicer* navíc postupně ukazuje vyhodnocení výrazů),
 - **Step Out** – vynoří se z volání funkce,
 - **Resume Program** – běží až po další breakpoint.

Čtení chybových hlášek

- Chyby při spuštění.
- Chyby za běhu.

Traceback (most recent call last):

File "locals.py", line 16, in <module>

result = outer(9)

File "locals.py", line 10, in outer

y = inner(n + 7)

File "locals.py", line 5, in inner

return y + 11

NameError: name 'y' is not defined

- Kde je problém? (Funkce, číslo řádku.)
- Co je to za problém? (Typ chyby.)

SyntaxError (chyba při spuštění) špatná syntax: chybějící dvojtečka nebo závorka, záměna = a ==, ...

IndentationError (chyba při spuštění) špatné odsazení.

NameError špatné jméno proměnné: překlep v názvu, chybějící inicializace proměnné.

TypeError špatný typ pro danou operaci: sčítání čísla a řetězce, ...

IndexError chyba při indexování řetězce, seznamu apod. (uvidíme časem).

... a další.

Jak spočítáme největšího společného dělitele dvou přirozených čísel?

- Jak to udělat *efektivně*?